overload|75

October 2006
ISSN: 1354-3172
www.accu.org

Up Against the Barrier
Simon Sebright

C++ Unit Testing Easier: CUTE
Peter Sommerlad

Inventing a Mutex
George Shagov

From CVS to Subversion
Thomas Guest

OVERLOAD 75

October 2006

ISSN 1354-3172

Editor

Alan Griffiths
overload@accu.org

Contributing editor

Mark Radford
mark@twonline.co.uk

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Thaddeaus Froggley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@gmail.com

Pippa Hennessy
pip@oldbat.co.uk

Tim Penhey
tim@penhey.net

Advertising enquiries

ads@accu.org

Cover art

Pete Goodliffe

Design

Pete Goodliffe

Copy deadlines

All articles intended for publication in Overload 76 should be submitted to the editor by 1st November 2006 and for Overload 77 by 1st January 2007.

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

The ACCU

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

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

Life in the Fast Lane

The ISO Fast-Track is paved with good intentions, but does it lead where we want to be?

The latest in the ISO C++/CLI story

I'm going to come back to this after explaining the process of standardisation, but the recent fast-track ballot by ISO's SC22 committee on whether to adopt ECMA's C++/CLI proposal as a "Draft International Standard" decided "against". ECMA as the Fast Track submitter has been instructed to "decide whether they wish to schedule a disposition of comments meeting and to address all comments and update the text for a second DIS ballot".

I'm not sure what the other options are, but there is no reason to think that ECMA will do anything but press ahead with this standard – which, after all, is what their business is about.

ISO's "Fast Track"

ISO has traditionally developed standards internally based on principles of "consensus", "industry-wide" and "voluntary". Quoting from their website [ISO]:

> **How are ISO standards developed?**
>
> ISO standards are developed according to the following principles:
>
> **Consensus**
>
> The views of all interests are taken into account: manufacturers, vendors and users, consumer groups, testing laboratories, governments, engineering professions and research organizations.
>
> **Industry-wide**
>
> Global solutions to satisfy industries and customers worldwide.
>
> **Voluntary**
>
> International standardization is market-driven and therefore based on voluntary involvement of all interests in the market-place.

The same page gives the following overview of the process of developing a standard:

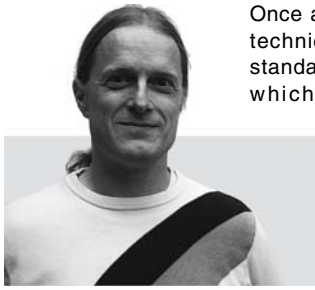
> The need for a standard is usually expressed by an industry sector, which communicates this need to a national member body. The latter proposes the new work item to ISO as a whole. Once the need for an International Standard has been recognized and formally agreed, the first phase involves definition of the technical scope of the future standard. This phase is usually carried out in working groups which comprise technical experts from countries interested in the subject matter.
>
> Once agreement has been reached on which technical aspects are to be covered in the standard, a second phase is entered during which countries negotiate the detailed specifications within the standard. This is the consensus-building phase.
>
> The final phase comprises the formal approval of the resulting draft International Standard (the acceptance criteria stipulate approval by two-thirds of the ISO members that have participated actively in the standards development process, and approval by 75 % of all members that vote), following which the agreed text is published as an ISO International Standard.

This is inherently a slow process – and, in practice, it can be *really* slow (in the case of C++ it took about nine years). And many feel that standards need to be produced on a more ambitious timetable. And so, ISO created an alternative "fast-track" process that allowed it to adopt standards created outside the above process – for example "publicly available standards".

While I can't find details of this fast-track process on the ISO website (having asked around, it is part of the "ISO/IEC JTC 1 Directives" – JTC is "Joint Technical Committee"). While I can't claim expert knowledge of these I understand that the fast-track proceeds as follows: firstly, document (for example a "publicly available standard") is submitted for "Fast-Track" approval to the corresponding ISO committee. In the case of the C++/CLI proposal this is SC22 (programming languages). This committee, or rather the national bodies that comprise it, then has a thirty day period to identify contradictions that would prevent the standard being considered. (We'll revisit this idea of "contradictions" in the context of the C++/CLI submission below.) Assuming the standard survives scrutiny, the next stage is for the committee to vote on its adoption as a "draft international standard". (This is in contrast with the process outlined above where a working group formed from the national bodies on the committee work together to create a consensus before the corresponding vote.)

The progress of C++/CLI (described below) suggests that the criteria for accepting a standard for fast-tracking are not clear and that there is a failure to gain consensus from people knowledgeable about the subject area and potential impact of this standard. There is a marked contrast with standards developed by ISO itself – for these there is, for example, a requirement that a working group comprising at least five national bodies is willing to take responsibility for the work involved.

Every organisation develops a culture over the course of time that reflects the way it tries to work. And ISO is no exception to this. Most of its standards are of interest to a minority of those on the decision making panels (not surprising really, there are a lot of standards and very limited resources to pursue them). A consequence of this is that national bodies with no interest in a particular standard try to keep out of the way by



Alan Griffiths is an independent software developer who has been using "Agile Methods" since before they were called "Agile", has been using C++ since before there was a standard, has been using Java since before it went server-side and is still interested in learning new stuff. His homepage is http://www.octopull.demon.co.uk and he can be contacted at overload@accu.org

automatically voting "approve" to anything that comes up for a vote. For traditional standards this is justified on both the tit-for-tat principle that others will do the same for standards that do interest them and on the assumption that those national bodies forming the working group have been diligent.

In organisational terms the fast-track is a new thing and the existing culture of "approve" by default is still operating. However, for a PAS submission there may be **no** national bodies interested in the standard – with the consequence that neither the tit-for-tat principle nor the presumption of diligence need apply. The effect of this can be quite alarming.

What does this mean for C++/CLI?

When it came to ECMA C++/CLI standardisation effort there was very little existing interest in SC22. Even within SC22/WG21 (C++) it was very much a minority interest – although some members of the BSI panel (and other participants in WG21) had made an effort to engage in the ECMA process. The general attitude, however, of those working on ISO C++ was that C++/CLI would be a diversion of their effort from more important issues.

When C++/CLI was submitted for "Fast-Track" the BSI C++ panel believed that there were a number of contradictions inherent in the C++/CLI submission (these are discussed in the editorial mentioned above). These were duly raised with SC22 which then chose to proceed with this submission. It isn't clear whether the objections raised by BSI, DIN (the German standards body) were considered inconsequential or irrelevant "technical comments". It does appear that the latter may be the case as, at this time, a document was posted to the BSI website requesting that national bodies "review and contributions on the relevant parts of JTC 1 Directives clause 13 on the 30 day call for contradictions during Fast Track" which suggests that it is recognised that there is ambiguity in this area.

It wasn't just the British and Germans that has issues to raise: the French C++ group also attempted to raise objections through their standards body (AFNOR) but I don't think these were actually seen by SC22 as there was some confusion regarding the language in which the objections were stated. My understanding is that the C++ group raised and/or translated their objections into French, thinking this was a requirement only to find that AFNOR would not submit them to SC22 unless they were in English.

Attitudes within WG21 changed somewhat during the Berlin WG21 meeting when, at an off-schedule get-together (sponsored – after the fact – by ACCU), Lois Goldthwaite presented the concerns of the BSI panel to many of the people active in WG21. One of the key points made in this presentation was that while C++/CLI was *a natural evolution of C++* (to the CLI) it could, if ratified by ISO, be mistaken for *the natural evolution of C++* that they had all been so busy working on. For ISO to ratify two incompatible "C++" standards in rapid succession would be guaranteed to create confusion.

As a result of these arguments the various national bodies active in WG21 chose to look at C++/CLI more closely and decided that they needed to act. This decision was not made lightly as there is an expectation within ISO that "no" votes are accompanied with "comments" (an informed explanation of the reasons for voting no and a suggested resolution that would change the vote to a "yes"). These comments are hard work – which discourages "no" votes on trivial grounds.

The consequence was that when it came to voting there were eleven votes in favour of accepting the C++/CLI standard and nine votes against. According to Francis Glassborow (who is better placed than I am to know who is active in WG21) all of the active members of WG21 voted against[1]: "No one who knew anything about the subject was in favour". While a majority of national bodies did approve the standard this represents a failure to gain approval (according to the voting rules there needs to be at least 66% approval and at most 25% against – abstentions not counting).

If you look at the numbers, they prove how hard it is to defeat one of these ballots. There were eleven in favour when twelve were needed to pass. And there were nine against when six would not have been enough. Consider what would happen with a working group that, like many, has fewer active members than WG21. Or one that was less alert to events.

1. There is a partial exception in the case of Switzerland – which national body voted "yes" without first consulting its C++ experts.

The role of ECMA

There are many bodies in the world that take it upon themselves to issue standards and ECMA (formerly the "European Computer Manufacturers Association") is one of many. As its former name suggests it represents the interests of manufactures (although, in line with current trends in globalisation, it in no way restricts its activities to Europe).

ECMA seems rather proud of the fact that it is behind the majority of standards fast-tracked through ISO since that process became available: "All together 250 standards have been fast-tracked since 1987:Over 80% of these come from Ecma" [ECMA]. Thus success progressing standards through the ISO fast track process is thought to encourage manufactures to invest in developing standards through ECMA.

The role of ECMA is different to that of ISO – the manufacturers that it represents are only one of the constituencies represented in ISO. The fact that, for example, Microsoft can gain consensus with other manufacturers for standardising C++/CLI is not a guarantee of more widespread support.

What does this mean?

There would appear to be something missing in the ISO fast-track process. While it does meet its initial goal of permitting standardisation on a much shorter time-scale it currently fails to meet the principle of consensus cited above.

There is a serious danger of the ISO stamp of approval being applied without due caution. One may hope that the review being undertaken by JTC1 will result in the process being updated to ensure that, for example, there is a quorum of national bodies interested in adopting any standard being considered.

Pending such a development there is a need for vigilance on the part of national bodies and, to assist them in this, on the part of individuals that may be aware of developments that their national bodies are not actively tracking.

Watch this space

There is another standard that bears watching to illuminate this process. Earlier this year ISO standardised Open Document Format (a format covering various office documents) through the Publicly Available Standard process when it was submitted by Oasis. Now ECMA is standardising Office Open – a corresponding format based on the new format that Microsoft's office suite will use. In due course this too will be submitted for fast tracking by ISO.

It seems absurd for ISO to standardise two competing standards in this space. This absurdity has a cost – the national bodies then have to publish these standards.

A note of thanks

I'd like to thank all those who responded to my appeal for help putting this issue of Overload together. In particular, I'd like to mention Alistair McDonald and Anthony Williams who both helped with the process of preparing material submitted by others for publication.

References

[ISO] http://www.iso.org/iso/en/stdsdevelopment/whowhenhow/how.html

[ECMA] http://www.ecma-international.org/activities/General/presentingecma.pdf

Letters

Colin Paul Gloster writes:

Dear William,

In response to Bill Fishburne's example comments for a stack in his article 'Comments Considered Good' in the August 2006 issue of *Overload* and his claim that "The comments offer us something that code alone cannot. There is no way for the declaration to state that size will decrease by 1 as a side-effect. [..] the code cannot document itself, particularly in side-effects", I really do not agree with this. Postconditions and the Design by Contract™ paradigm are capable of documenting this without a comment, e.g. SPARK [SPARK] and Eiffel [Meyer] can do so. (SPARK is defined in such a way that it is entirely written inside comments of another language's. This does not affect my point that postconditions in SPARK are documented without a comment: the commenting is for bypassing the other language's compiler, before the other language's compiler is invoked the SPARK code must be passed by a SPARK analyzer.)

Regards,
Colin Paul Gloster (Colin_Paul_Gloster@ACM.org)

[Meyer] Bertrand Meyer, 'Part 6 Design by Contract and Assertions' of 'Invitation to Eiffel', HTTP://Docs.Eiffel.com/eiffelstudio/general/guided_tour/language/invitation-07.html

[SPARK] Praxis High Integrity Systems, 'SPARK Quick Reference 1 - Toolset and Annotations', WWW.Praxis-HIS.com/sparkada/pdfs/SPARK_QRG1.pdf

William Fishburne replies:

Colin,

Thank you for your comment. I was speaking particularly in respect to C/C++, and am sadly unfamiliar with both SPARK and Eiffel (I know Eiffel by name only). It does seem likely, however, that any language which could completely forego comments would be so verbose as to rival a native language (such as English), although I would surely not want to bet the farm on it! One might argue that 'side-effects' would be impossible in such a language, but I will let my ignorance of the two languages in question strongly undermine the validity of any such argument from me.

Instead, let me direct you to http://www.ddj.com/184405997 which is an article by Christopher Diggins who built Heron using the Design by Contract paradigm and stated (with respect to the paradigm and not specifically to Heron):

> In fact, some pre/postconditions are difficult and even impossible to express in a formal language. In these cases, the pre/postcondition is best expressed as a natural language comment.

I'm unable to evaluate the validity of this assertion, but simply point you toward it as it seemed relevant to me.

William Fishburne (bfishburne@gmail.com)

Colin Paul Gloster again:

Dear William,

I do not know of any programming language in which comments are not possible, but I can think of at least one terse specification language (namely Z) which does not have comments. The objection I have written is not against the overall idea that comments may be good, but rather against a specific example in which it had been claimed that code visible to the user other than a comment can not document a particular feature.

I agree it [the Christopher Diggins quote] seems relevant. Not everything is expressible in every or maybe even some language, and a comment may be helpful to explain something complicated which is written or which can not be written in a formal language.

Regards,
Colin Paul Gloster

Seweryn Habdank-Wojewódzki writes:

Dear Rachel Davies,

I read your article about Pair Programming [Overload 73]. Last month I made a survey of software projects. I would like to share my observations with you.

The survey was of 48 software projects. A questionnaire was completed, recording metrics such as the number of lines of code, the number of developers, the number of different languages and APIs used, and so on. A copy of the questionnaire can be made available if desired.

In figure 1, I show cloud data which shows a correlation between complexity of the project and the number of developers.

This shows the trivial fact that as complexity increases, the number of developers increases. We could also turn this round and suggest that more developers can spontaneously increase the complexity of a project.

One other interpretation we can make is how much complexity can the average developer cope with? This appears to be around 1 developer for 5 points of complexity. This might be useful for managers in estimating how many developers are required for any particular project.

Figure 2 shows the number of lines of code per person month, plotted against the number of developers on a project. Looking at the top of the graph, we can observe that there is small tendency for developers to create

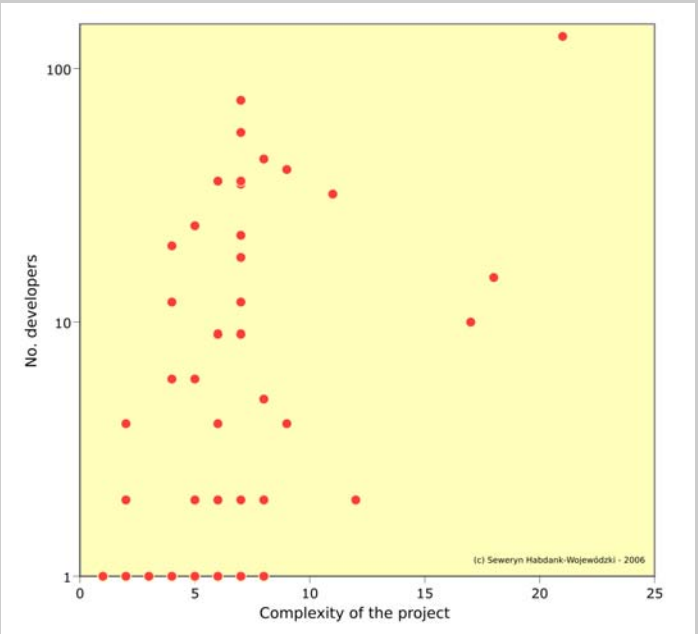


Figure 1

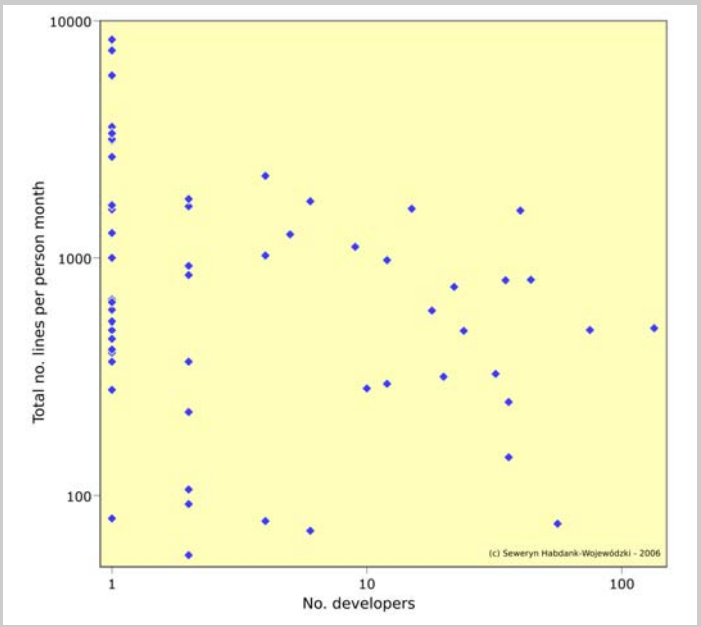


Figure 2

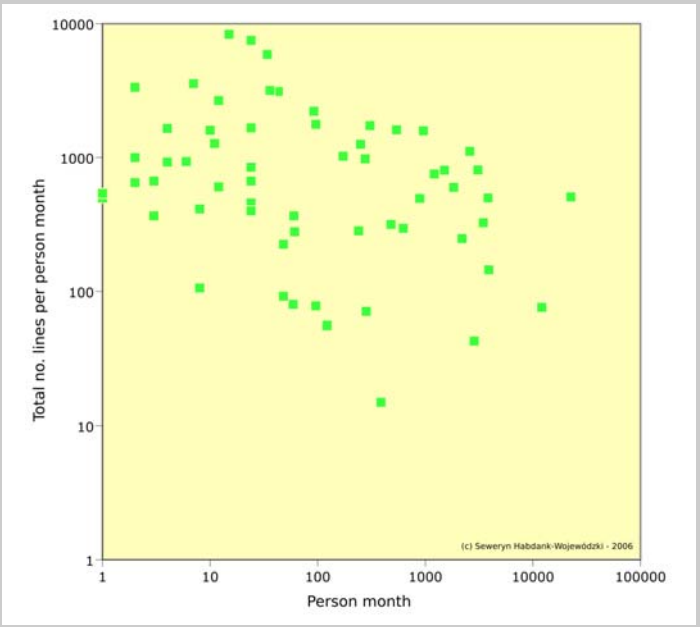


Figure 3

less code as the numbers of developers in the project increases. On the other hand, looking at the bottom, the least productive developers create more code as the number of developers increases.

This may be because, as team size increases, developers spend more time on communication between themselves, but conversely they are stimulating each other to increase speed, possibly because a group of people is more creative than the individuals alone.

Figure 3 shows the number of lines of code per person month against the duration of the project in person months. It shows that coding speed decreases on a longer project.

There may be two reasons. Firstly, to get a high number of person months, there may be many people on a relatively short-running project. Many people need more time to communicate, instead of coding. Secondly, a small group of people may work on a long-running project, develop a large codebase, and they cannot cope with it, perhaps through boredom or tiredness. This would suggest that programmers are better when they are new in the project, but of course there is a low boundary, as it takes time for a developer to attain good productivity.

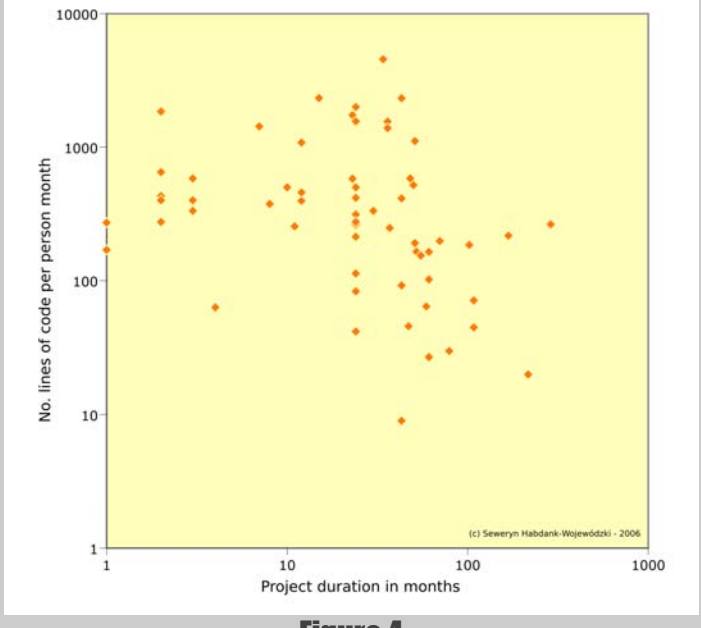


Figure 4

Figure 4 shows number of lines of code per person month against project duration (only data for projects between 20 and 30 months duration are shown). As can be seen, as project duration increases, coding speed slows.

My question is how do you think Pair Programming would affect the results – would the same trends be seen, or would pair programming projects correspond to the same graphs?

Best regards,
Seweryn Habdank-Wojewódzki (habdank@megapolis.pl)

Rachel replies:

Dear Seweryn,

I apologize for taking so long to reply to your questions. I have looked through your observations and the graphs are interesting.

Your conclusions on how much complexity a developer can handle make the assumption that the project did deliver/complete and that developer competence is a constant. Your survey did not ask is what calibre of developers were used and also whether the project delivered to completion. In my view, the graphs only tell us is the level of staffing that the organisation felt was appropriate, this might depend on other factors not just complexity (such as how much money is available, expected working hours, number of developers already in department, etc.). Also, I am also not sure that having a coding standard or doing testing (questions 11 and 12) increase the complexity of the project.

Regarding the other pictures (developers-lines of code), I would add that I do not expect lines of code to keep growing at the same rate throughout the project. I would hope that the team practice refactoring and have an on-going effort to reduce code bloat in favour of cleaned up designs. In this study, we don't have information on other factors that could be affecting developer productivity (company events, time spent in meetings, etc). I don't think lines of code is enough to assess developer productivity. I think you need to combine information about code quality and customer satisfaction to understand if these developers were truly productive.

I am not good a speculating but would hope that pair programming would improve code quality and if that were factored into your measure of productivity then pair programming ought to improve productivity but have a slower rate of adding code due to more emphasis on refactoring.

Best regards,
Rachel Davies

Up Against the Barrier

A discussion of development against the odds, describing process, personal and environmental barriers between the people and the job they are trying to do.

This article seeks to explore why things are not always done 'properly'. Based on personal experience working in various teams, I have noticed a common problem: some people in the team 'know' what is idiomatic, what good and bad practices are, and yet the code still rots. I would like to explore with you why this might be, and thus draw some conclusions for both the software developers and the project managers.

Fundamentally, I have identified that there are various 'barriers' in place when a developer is given a task. These barriers can be to do with the process, the tools in use, the other people on the project, or the developer themselves. Typically these barriers are of the kind that narrow the amount a developer touches a code base, and I will argue that this is often just the opposite of what is needed.

The situation where barriers exist tends to encourage practices which I identify as anti-patterns. That is, they are things which occurred frequently, in response to the forces in play, but because the forces in this case were the barriers hampering good practice, the result was a degradation of the code base.

The most significant barrier I encountered was the disabling of multiple checkouts in the source control system. By this, I mean the ability for more than one developer to checkout a given file simultaneously, such that the can all make changes independently.

Let's explore what the barriers might be, with an attempt to categorise them:

Process

- Very bureaucratic change process
- Poor access to files e.g. multiple checkouts not allowed, remote working connection speed problems
- Restrictions on code changes. E.g. in later phases, management decide to touch as little as possible
- Lack of automated testing

Personal

- Lack of ability to make the right decisions about changes
- Fear of failure
- Wanting to keep a low profile
- Laziness

Environment

- Long build times
- High dependencies among modules

I decided to write this article based on a small subset of these barriers, which I had identified as particularly prevalent in one place of work. However, thinking about the topic, I identified other barriers too, which might be causing similar problems for other teams.

So, the talented developer can often be frustrated at the system, his colleagues or his manager. The question I think is important is: why are these barriers a problem? What effect do they have exactly?

Let's examine the kinds of changes that are made to a code base once the project is well under way, since these are often different to the changes made in the opening or closing stages of the project, and this is when the majority of the changes are made. There is compilable code which correctly implements some desired features. There is more functionality to add, existing functionality to change as requirements change, and there are bugs to find in what exists. Thus, changes are afoot, rife perhaps, to existing code. We are actually in maintenance mode as well as new development mode. Assumptions will change and some new features will pose challenges to the designs in place.

So, let's consider a developer who is assigned a task. The product has a graphical front end and works with a hierarchy of objects. We display the name of the item in the list view, but for some items, we wish to display something else. Perhaps a different kind of name, for example a name embellished with other information like the count of sub-objects. The experienced among us might immediately pull out new concepts from this. Aha, we have the concept of displayed name, or list view display value, or display value parameterised by view type. We might have a session at the whiteboard and/or pub to see which concept best fits the expected future changes (perhaps other views might want to display different information too).

Our programmer, however, does not think like that, or at least does not act like that. They want to have an easy life, sign off the requirement and go to the pub. What is the easiest/quickest way to get this done?

Answer: Go to the list view code, and where the name is being pulled off the items, do a dynamic cast to see if we have the special type. If so, call a function on it to get the extra information to display and append this to its name. One function, plus a `#include` statement: one source file changed. Ah, and no one has that file checked out. Magic!

Someone else given this task would have approached it differently. They would have introduced the concept of different display properties, perhaps parameterised by view type, or something. Ok, so we add some new virtual functions to the object base class. Provide a suitable implementation for most items, and for the special item, we provide the desired implementation. Maybe all the view classes need to be given knowledge of this new concept too. Without getting into a discussion about the merits of implementations in virtual function hierarchies, etc., I hope you can see where I am coming from. The changes necessary here are now much wider, in that they affect both more files directly through code changes, and indirectly through dependencies, most notably the object hierarchy. We may even end up with a fundamental change to both the objects and the view. Heaven forbid!

What have I explained here? That the changes you can make to software can be done in different ways. I think there is often a correlation between

Simon Sebright has been programming for 10 years, mainly in multi-tier C++ application development. Recently, he has been designing and developing web/database-based applications using C# and asp.net. He can be contacted at simonsebright@hotmail.com

the correctness of a change and the amount it affects the code base. Here is a hierarchy of changes, ordered by ascending effect on dependent files:

1. Code change in one function, as outlined above. Note the nasty smell of a dynamic cast, or other form of type ascertainment.
2. Add a parameter with a default value to a function that needs to do something different in some circumstances, and call it with a non-default value where the new behaviour is required. (requires change to two files for the changed class and files for calling classes where some different behaviour is required, and rebuild of dependent things).
3. Add a new function (same overhead as above, but with additional potential responsibilities of documentation, etc. and the visibility of having changed that interface more).
4. Add a new virtual function to a base class (requires access to the base class, and the derived class for which the specialised version is required, requiring access to more files and a longer rebuild). Perhaps other families of objects need to change too to know of the new concept in town.

In the team where this kind of thing was most prevalent, there were two main barriers in play. One was personal: laziness, lack of thought and ability to pull out concepts. The other was process – multiple checkouts were not allowed. As a result, the codebase was already in decline a long time before the end of the project. Particularly prevalent were functions which had a Boolean parameter at the end, one of the lower impact changes mentioned above. This was my anti-pattern nemesis.

Anti-Pattern – Parameterise Function with Boolean (often with default value)

Forces: Some functionality exists, but a change in behaviour is needed in certain circumstances. Other places using that functionality are required to remain the same.

Solution: Add a Boolean parameter to the function. Give it a default value, and in the function body, preserve the current flow through the code for that value. For the non-default value, do something different as required.

The parameter would often be called doX or dontDoX, indicating that you could get it to do something subtly different with either true or false. There would often be a default so that existing code would compile and do the same as it did. Most people reading this article would try and think of more graceful ways to solve this problem; use of Template Method design pattern perhaps, or perhaps two separate functions, each of which calls a helper to do the common bits, the names of these functions clearly indicating which behaviour they cause. The point is that that process of thinking simply did not occur. Sometimes it was lack of ability or experience, sometime laziness.

Consequences: The results of this anti-pattern are mysterious calls to functions. `GetName( false );` Whilst totally comprehensible to the developer at the time, should either the developer or the time change, comprehensibility declines. One could use techniques designed for comment-less coding to mitigate the mess somewhat, such as declaring a named constant variable and passing this in, or using an enum, but I feel this is really tidying up after a mess, rather than avoiding creating that mess.

But there were code reviews; surely only good code will get checked in?

At least, the process included a mandatory code review for each checkin. The reviewer had to check a box on the requirement sign-off page. Unfortunately, this review more often than not ended up as simply reading the code. Any questions asked were dismissed with lame excuses. “Oh, yes, I could do it that way, but this works and is much simpler.” “That would be overkill.” “The files were checked out.”

I managed to get a reputation for being a difficult reviewer, because I would demand changes where I felt that things were not being done properly. A few people in the team really liked this, and we formed a band offering good constructive criticism of each other’s designs and code. Other team members sought reviewers of the old school type so they didn’t have much hassle.

Now for the second main barrier in place: the lack of multiple checkouts. This was rigidly enforced and defended by the project management team for fear of collisions of code changes causing compilation failures and bugs. It is certainly true that when checking in a file that someone has changed under your feet, you have to go through a process of reconciling those changes and making sure your own changes still make sense. But, that is something you have to do anyway when making a checkin, because the same issue occurs between different source files. For example, the same issue still applies if you decided to call a function which no longer exists on a class, or for which someone has changed the signature.

So, their worry was really not fully founded in my opinion.

constraints you impose on your team must be those which will encourage better code being checked in

Furthermore, it had exactly the opposite effect to that intended. The more diligent programmers did try to do things properly and, as discussed above, often needed to make small changes to a larger number of files. We often ended up in deadlock. Because someone knows they won’t be able to checkout a file if someone else grabs it first, they check it out as soon as they can and hog it. So, the second person needing it does the only sane thing and makes their own copy writeable so as they can get on with their own changes. There are ways to make this more robust. These might range from simply writing down which files are thus affected, to making a temporary source code control system for such situations. However it is done, we have the same risks as when the main tool is used to do multiple checkouts, but with extra overhead and risks as well! These extra risks are things like the developer accidentally overwriting their working copy, or forgetting to integrate the changes made in the meantime. The multiple checkout facility, in this case of SourceSafe, is specifically there to help manage this situation for the developers, but it wasn’t being used, for fear of the very thing which it helps to avoid.

Now, it also had the effect of changing the nature of changes made to the files. Things tended to back up on developers’ machines who were not lucky enough to get all the files they needed, thus changes were often much larger than they could have been. Several requirements might be addressed in one checkin, for example. With multiple checkouts allowed, this would not be the case, as you have the ability to keep checking in your atomic changes, and keeping them concerned with one issue at a time. So, we had a vicious circle. Lack of multiple checkouts made the lack of multiple checkouts much worse to live with!

What can we learn?

Any constraints on developers will have a tendency to reduce their effectiveness, when considering the project as a whole, evolving thing. Constraints can be those imposed by the management via process, or the tools in use, through to the developers themselves, through laziness, lack of ability, or lack of experience.

If you are a project manager, be aware that the constraints you impose on your team must be those which will encourage better code being checked in, not merely conform to some loose idea of how things should be done. A good constraint might be “any code checked in must have unit tests”. A bad constraint might restrict access to files (as with the well-intended lack of multiple checkouts).

If you are a developer, think hard about what you check in. Is it a result of some compromise because of file availability, or because you didn't feel like changing base object class for some reason? Think how your changes will affect the future of the code, and if you can make that future brighter, then consider doing so. Fight your corner where you can see that well-meant processes are causing a derogatory effect.

There will be barriers in place at your place. Hopefully they will be lower than you can jump. ■

Inventing a Mutex

A mutex is a general purpose tool – there may be better solutions in specific circumstances. George Shagov presents one such alternative.

Introduction

The article might be of interest for server developers in a multi-user multi-threaded multi-processor environment with high data flow capacity, where performance stands on the very edge of the business. (Note that the approach presented here is restricted to systems where the application runs on a single core, and that extreme care should be taken applying it to further processor architectures. - Ed) The basic objective is to represent another approach to the locking mechanism, well-known as mutex objects. Sometimes synchronization objects are the very edge of performance, in some systems locking might take up to 30 percent of total performance, which is quite annoying, since in order to improve the performance it is required to change the architecture of the system, if it is possible of cause, sometimes for certain reasons it is not.

Classical mutex

I do not want to spend your time explaining how mutexes work, yet I think it is important to refresh this knowledge in order to understand the basic idea of the article, and all the consequences which follow. The realization of the synchronization object basically depends on the CPU architecture, let us consider an Intel x86 based one. Basically the mutex object itself is but an integer value (if we are talking about non re-entreat objects). The mechanism of obtaining the lock is usually (this is a very general assumption) performed by means of the assembler **`XCHG`** instruction, which exchanges two values. So if we set **`EAX`** register to **`0`** and will perform **`XCHG EAX`**, **`DWORD PTR [EBP]`**, where **`EBP`** points out to the mutex (which is integer), all we need to do after is to check the **`EAX`** value, (**`0`** represents a lock with **`1`** being a fail). The crucial thing I did not mention is the **`LOCK`** prefix that must be used before **`XCHG`** instruction, which locks the **`BUS`** during **`XCHG`**. Actually that's it. It works perfectly fine, the basic problem is this prefix **`LOCK`**, since locking the **`BUS`** is quite a costly operation from the perspective of view of performance.

Surprise

What we can do in order to make this operation lighter is to merely remove the LOCK prefix before the XCHG instruction. And that will work, since XCHG is an atomic operation and we assume we are talking about a single CPU system; in a multiple CPU environment, it will definitely fail.. This limitation does look strong, and so it is, but in some cases it might help, in particular where I am describing 'Limitation and usage' in the section below.

It may seem that we could remove the LOCK prefix, but on closer examination this would gain nothing:

> The XCHG (exchange) instruction swaps the contents of two operands. This instruction takes the place of three MOV instructions and does not require a temporary location to save the contents of one operand location while the other is being loaded. When a memory operand is used with the XCHG instruction, the processor's LOCK signal is automatically asserted. This instruction is thus useful for implementing semaphores or similar data structures for process synchronization. See "Bus Locking" in Chapter 7, "Multiple-Processor Management" of the IA-32 Intel® Architecture Software Developer's Manual, Volume 3A, for more information on bus locking."[Intel06]

The AMD has the same.

I do not know for what reason they have done that, but that is the fact, probably it means some kind of a protection from a fool, I honestly do not know. So, it might seem like this is a dead end. Actually, it is not.

Logical lock

Let us say we do limit the amount of the users by means of some prior known value, for instance 32. Let us say that our synchronization object will be 32-bit value, an integer, a bit mask, let us say that each particular bit of this mask corresponds to one particular customer (a thread), let us say each thread has and knows its id, let us say each thread is allowed to change only the corresponded bit in the bit-mask, and not allowed to modify any other bit, yet is allowed to check any bit in the bit-mask. Have you got the point? As simple as that, the algorithm of getting lock for thread with number K in case of non re-enter able synchronization object might look like figure 1. The steps are described on the next page.

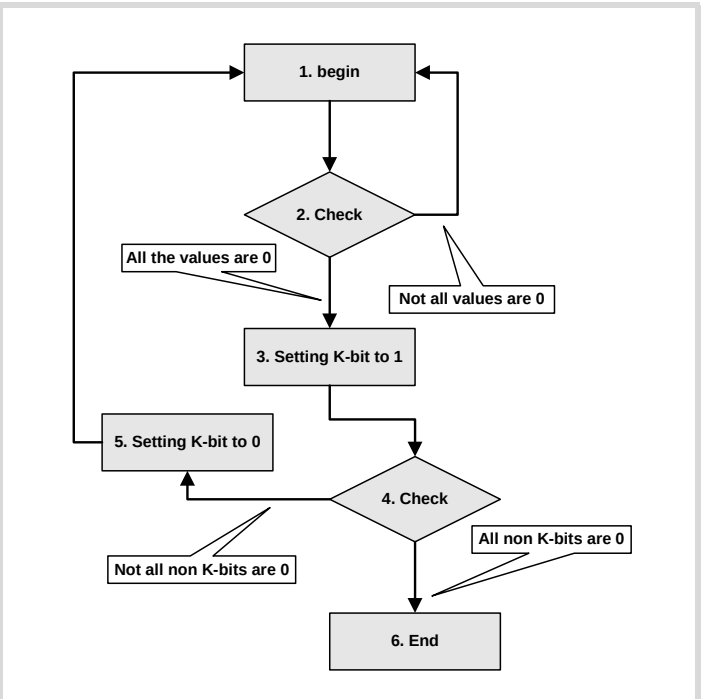


Figure 1

George Shagov was born in Moscow in the seventies. Graduated from Moscow State Technical University. He started as a software developer in 1992 and since has spent a year and a half in the US (also developing software). George is currently working as a software engineer at Deutche Bank in Moscow. George can be contacted at george.shagov@db.com

Sometimes synchronization objects are the very edge of performance, in some systems locking might take up to 30 percent of total performance

1. The beginning of the algorithm.
2. Checking that all the values of the bit mask are 0. (This step might have been omitted). And here we have two options:
   - All the values are 0. It means the mutex is in non-signalled state and we are allowed to get a lock
   - Not all the values are 0. The mutex is in signalled state, some thread has got the lock already. This is the very time to perform spinning and then to go to the beginning of the algorithm.
3. Customer with number K is allowed to modify only K-th bit in the bit mask. On this step the modification is performed.
4. It might have happen some other customer has got the lock during this time (from step 2 to 4) therefore we need an additional check here. We need to check that all non-K bits in the bit mask are 0, there are two options also:
   - All non-K bits in the bit mask are 0, it means we have got the lock.
   - Not all non-K bits in the bit mask are 0, it means we must reset K-th bit in the bit mask to 0 (step 5) and go to the beginning of the algorithm.

The code is absolutely simple and to post it here would be the waste of the space, the only think to consider is the third and fifth step, which are to be done in assembler, my version is in Listing 1.

C interface for these functions:

```
extern void nl_nr32_set(
    volatile word32* synchronisation_mask,
    word32 thread_bit);

extern void nl_nr32_reset(
    volatile word32* synchronisation_mask,
    word32 thread_bit);
```

Does it work? Approximately four times faster than classical mutex. Which is quite obvious since the value of the mutex gets cached. Therefore we must say:

Once again

Generally it does not work, since in general case we have a multiple CPUs and each particular CPU has its own cache, therefore in multi-processor environment the algorithm will definitely fail.

It means in order to have got this working we must place all the threads in one CPU, which looks quite strong limitation.

Some data

The first test (see Table 1) creates a thread which locks the mutex and unlocks the mutex, and thus it does for one minute, the amount of locks/unlock is outputted.

```
/*
 *  set (step 3)
 */
.globl nl_nr32_set
  .type  nl_nr32_set, @function
nl_nr32_set:
  pushl  %ebp
  movl  %esp, %ebp
  movl  8(%ebp), %edx
  movl  12(%ebp), %eax
  orl  %eax, (%edx)
  popl  %ebp
  ret
  .size  nl_nr32_set, .-nl_nr32_set
/*
 * reset (step 5)
 */
.globl nl_nr32_reset
  .type  nl_nr32_reset, @function
nl_nr32_reset:
  pushl  %ebp
  movl  %esp, %ebp
  movl  8(%ebp), %edx
  movl  12(%ebp), %eax
  and  %eax, (%edx)
  popl  %ebp
  ret
  .size  nl_nr32_reset, .-nl_nr32_reset
```

Listing 1

The second test (see Table 2) creates 8 threads and each one locks the mutex and unlock it, for one minute, the amount of locks/unlock is outputted.

Such poor number of locking count under Windows I can not explain, this is a question rather to be asked to Microsoft.

There are something to consider here, videlicet a spin-time. On some systems (let me say thus) it significantly changes the result. I am not going to analyse the matter; the only one thing I intend to say is that applying such a lock needs to be diligently considered and measured in each particular case and all the benefits and disadvantages must be taken in account. Such a lock is a very tricky way to get the performance and applying such a one without understanding might lead to unpredictable results.

Limitation and usage

The limited amount of threads and requirement to have all these threads executed on one CPU are to be considered as strong limitations. Notwithstanding there are some certain areas where this invention might have been successfully used, for instance:

Mandriva 2006. Intel Celeron 2.4Ghz	Suse 9 Intel Xeon DP 2.8GHz	Windows XP/Cygwin Intel Xeon(TM) 2.8GHz
./nlocks_test_01 p_thread started count: **315107638** p_thread finished nl_nr32_thread (within pthread_self) started count: **1453393599** nl_nr32_thread finished	./nlocks_test_01 p_thread started count: **400931049** p_thread finished nl_nr32_thread (within pthread_self) started count: **1800204642** nl_nr32_thread finished	$ ./nlocks_test_01.exe p_thread started count: **61242829** p_thread finished nl_nr32_thread (within pthread_self) started count: **145314858** nl_nr32_thread finished

Table 1

- Let us consider some multi-threaded business server, which has some kind of the data collection which is intensively used by means of one or two threads, one thread for instance provides the pretty high flow of the incoming data which are supposed to be added at the collection, the second performs some kind of a calculation, and probably removal of the data from the collection and also there are some amount of threads which lazily query the collection. I would presume the task is quite common. In this case these two threads, which have a high flow of the data might use the locking mechanism, described above. It is important to remember these two threads are to be started on one CPU, moreover we need an additional stuff of threads which are to be started on the same CPU which will perform query operation. This approach will cause some performance degradation for querying threads, yet it will boost up the performance of the first two threads. Thus it becomes possible to get the benefit.
- Let us consider a game server, which offers a skirmish service. It is quite clear what must be done here. All the members (their threads) of one game must be placed on one CPU (though we do limit their amount), and thus we swerve from common locking mechanism. Unfortunately I can not tell you how much percentage of the performance might be taken here, I tried to contact the companies who offer such services but have not had an answer.
- Here I must mention different chat and conference virtual rooms, the only problem is the limited amount of the users.
- I can not avoid the mention of PC boxes, most of them have only one CPU, therefore in some cases classical locks might be considered as redundant.
- One more interesting thing to consider. The algorithm must perfectly on in case multi-core processors, if they use one cache.

In conclusion

I must say the know-how must be diligently considered before use and additional investigation must be done on each particular case, which seems to be quite heavy to apply. Notwithstanding, this is a way to go for those developers who deal with the application where performance is a key issue, such as real-time systems. ■

References

[Intel06] IA-32 Intel® Architecture Software Developer's Manual. Volume 1: Basic Architecture. Order Number: 253665-018. January 2006

Mandriva 2006. Intel Celeron 2.4Ghz	Windows XP/Cygwin Intel Xeon(TM) 2.8GHz
./nlocks_test_02 ---------- Initial value: 99 p_thread 3084577712 started <skipping> p_thread 3025828784 finished <skipping> Thread id: 3084577712[0], counts: 25509563 [*1] <skipping> Sum: **304813761** [*1] Initial value: 99 ---------- Initial value: 99 nl_nr32_thread 3017436080[0] started <skipping> nl_nr32_thread 2967079856[6] finished <skipping> Thread id: 3017436080[0], counts: 84948965 [*1] Thread id: 3009043376[1], counts: <skipping> Sum: **715733620** [*1] Initial value: 99 Test finished	$ ./nlocks_test_02.exe ---------- Initial value: 99 p_thread 268502720 started <skipping> p_thread 268503040 finished <skipping> Thread id: 268502720[0], counts: 2930005 [*1] <skipping> Sum: **21867838** [*1] Initial value: 99 ---------- Initial value: 99 nl_nr32_thread 268503856[0] started <skipping> nl_nr32_thread 268504400[4] finished <skipping> Thread id: 268503856[0], counts: 50639884 [*1] <skipping> Sum: **349357548** [*1] Initial value: 99 Test finished

Table 2

C++ Unit Testing Easier: CUTE

Peter Sommerlad presents a lightweight framework for C++ unit testing.

This article describes my attempt to leverage more modern C++ libraries and features to make the writing of unit tests easier in C++. For example, one disadvantage of CPPUnit is that you have to write a subclass to have your own test case. This is a lot of programmer overhead, especially when you want to start small.

I was inspired by Kevlin Henney's Java testing framework called JUTLAND (Java Unit Testing: Light, Adaptable 'n' Discreet), and the corresponding presentation he gave at JAOO 2005. In addition, I wondered if I could come up with a design that is similarly orthogonal, easily extendable and also much simpler to use than current C++ unit testing approaches.

You will learn how to use CUTE in your projects and also some of the more modern C++ coding techniques employed for its implementation. I also ask you to give me feedback to further simplify and improve CUTE. Even during the writing of this article I recognized simplification potential and refactored CUTE accordingly. Porting and testing onto other compilers not available to me is also an appreciated contribution.

My problems with CPPUnit

Inheritance is a very strong coupling between classes. Requiring a test-case class to inherit from a CPPUnit-framework class couples both closely together. The CPPUnit tutorial [CPPUnitCookbook] lists at least six classes you have to deal with to get things up and running. You even have to decide if you want to inherit from **`TestCase`** or **`TestFixture`** for your simple test case you intend to write. I do not want to go into more details, but I show how I would like to write tests.

```
#include "cute.h"
int lifeTheUniverseAndEverything = 42;

void mysimpletest(){
    t_assert(lifeTheUniverseAndEverything == 6*7);
}
```

That's it. A simple test is a simple **`void`** function. Done (almost).

In addition CPPUnit stems from a time of non-standard C++ compilers, where more modern features of the language just have not been available broadly. This limits its design from a modern perspective.

Today, all relevant compilers (with the exception of the still in use MSVC6) are able to compile most of standard C++ and the proposed **`std::tr1`** libraries from Boost.

Running a single test

Since we lack some of the reflection mechanisms available in Java, you have to write your own main function for testing with CUTE. For simple cases this is straightforward. As with CPPUnit you have to instantiate a runner object and pass your test for running it. The simplest possible way, producing some output is shown in Listing 1.

OK, not very impressing yet, but simple. Reporting test outcome is so common, that CUTE provides a means to configure the runner with a so-called 'listener'. You already might have wondered what these template-brackets with **`cute::runner`** were about. So we can change that to the following:

```
#include <iostream>
#include "cute_runner.h"
int main(){
  using namespace std;

  if (cute::runner<>()(mysimpletest)){
    cout << "OK" << endl;
  } else {
    cout << "failed" << endl;
  }
}
```

Listing 1

```
int main(){
  using namespace std;
  cute::runner<cute::ostream_listener>()(
   mysimpletest);
}
```

```
"void () () OK"
```

is what we get from that one. We know that a **`void`** function without any arguments succeeded to run. This shows C++ introspection limitation, which only provides type information, not function names. But the preprocessor can help us. So making our test cuter by applying the macro **`CUTE()`**, helps us:

```
 cute::runner<cute::ostream_listener>()(
   CUTE(mysimpletest));
```

This achieves the output **`"mysimpletest OK"`**. Not too interesting. However, if we make our test case fail by setting deep thought's answer to 41 instead of 42, we get:

```
../simpletest.cpp:6: testcase failed:
 lifeTheUniverseAndEverything == 6*7 in mysimpletest
```

That is everything we need to track down the failure's origin and even some context helping with a first guess. You already guessed that the preprocessor **`macro t_assert()`** from `cute.h` contains the magic for collecting this interesting information.

Note that all classes presented in the following are in namespace **`cute`**, that I omit from the definitions shown for brevity.

Peter Sommerlad is professor and head of Institute for Software at HSR Hochschule für Technik, Rapperswil. He can be contacted at peter.sommerlad@hsr.ch

I tried to create a **simple** orthogonal and thus **easy to extend** and adapt testing framework

How things work

I tried to create a simple orthogonal and thus easy to extend and adapt testing framework that stays easy to use. Exploiting C++ library features that are modern (boost) and will be part of the future standard (**std::tr1**) I could avoid some complexity for CUTE's users.

CUTE test

The core class representing tests uses **boost::function**, which will be **std::tr1::function**, when compilers support this designated extension, to store test functions. So any parameterless function or functor can be a test. In addition each **cute::test** has a name, so that it can be easier identified. That name is either given during construction or derived from a functor's typeid. The GNU g++ compiler requires to demangle that name given by the **type_info** object, where VC++ already provides a human readable **type_info::name()** result.

```
struct test{
  template <typename VoidFunctor>
  test(VoidFunctor const &t, std::string name =
           demangle(typeid(VoidFunctor).name()))
  :theTest(t),name_(name){}
  void operator()()const{ theTest(); }
  std::string name()const{ return name_;}
  static std::string demangle(char const *name);

private:
  boost::function<void()> theTest;
  std::string name_;
};
```

As you can see, there is no need to inherit from class **test**. The only thing I do not yet like, is the static function **demangle**, that needs to be implemented per compiler. I haven't found a better place for it in the framework yet.

For simple functions, or if you want to name your tests differently from the functor's type, you can use the following **CUTE()** macro.

```
#define CUTE(name) cute::test((name),#name)
```

The using of a template constructor allows you to use any kind of functor, that can be stored in a **boost::function<void()>**, that means a functor that doesn't take parameters. With **boost::bind()** you are able to construct those, even from functions, functors or member functions with parameters as shown below.

Sweet suites

Running a single test with **cute::runner** is not very interesting – you might call that function directly and check results. But having a larger collection of test cases and running them after every compile and on a build server after every check in, is what makes unit testing so powerful. So there is a need for running many tests at once.

But in contrast to other unit testing frameworks (including JUnit) I refrained from applying the Composite Design Pattern **!** [GoF] for implementing the container for these many test cases your project requires.

I love **Composite** and it is handy in many situations for tree structures, but it comes at a price of strong coupling by inheritance and lower cohesion in the base class, because of the need to support the composite class interface. The simplest solution I came up with is just using a **std::vector<cute::test>** as my representation for test suites. Instead of a hierarchy of suites, you just run a sequence of tests. When the tests run, the hierarchy doesn't play a role. You still can arrange your many tests in separate suites, but before you run them, you either concatenate the vectors or you run the suites in your **main** function separately through the runner.

For those of you who really want your test suites to be tests, CUTE provides a **suite_test** functor that will take a suite and run it through its call operator. But if any test of the suite in such a **suite_test** fails, the remaining tests won't be run.

To make it easy to fill your suite with your tests CUTE provides an overloaded **operator+=** that will append a test object to a suite. This idea is blatantly stolen from **boost::assign**, which I didn't use, because to my knowledge it didn't make it to **std::tr1** (yet?).

So this is all it takes to have test suites:

```
typedef std::vector<test> suite;
suite &operator+=(suite &left, suite const &right);
suite &operator+=(suite &left, test const &right);
```

Assertions and failures

A unit testing framework wouldn't be complete without a means to actually check something in a convenient way. One principle of testing is to fail fast, so any failed test assertion will abort the current test and signal that failure to the top-level runner. You might already have guessed that throwing an exception is the corresponding mechanism. Since we want to know later on, where that test failed, I introduced an exception class **cute_exception** that takes the filename and line number of the source position. Java can do that automatically for exceptions, but as C++ programmers we have to carry that information ourselves and we have to rely on the preprocessor to actually know where we are in the code. Another **std::string** allows sending additional information from the test programmer to the debugger of a failing test.

This is how `cute.h` looks without the necessary **#include** guards and **#include of <string>**:

```
namespace cute{
struct cute_exception {
  std::string reason;
  std::string filename;
  int lineno;
  cute_exception(std::string const &r,
                 char const *f, int line)
  :reason(r),filename(f),lineno(line)
  {   }
  std::string what() const ;
};
}
```

One principle of testing is to **fail fast**, so any failed test assertion will **abort the current test** and signal that failure to the top-level runner

For actually writing test assertions I provided macros that will throw, it a test fails. I deliberately used lower case spelling for these macros to make them easier to use.

```
#define t_assertm(msg,cond) if (!(cond)) \
  throw cute::cute_exception((msg),__FILE__,__LINE__)
#define t_assert(cond) t_assertm(#cond,cond)
#define t_fail() t_assertm("fail()",false)
#define t_failm(msg) t_assertm(msg,false)
```

This is all for you to get started. However, some convenience is popular in testing frameworks. But convenience often tends to be over-engineered and I am not yet sure if the convenience functionality I provided is yet simple enough. Therefore I ask for your feedback on how to make things simpler or encouragement that it is already simple enough.

Equality - overengineered?

Testing two values for equality is may be one of the most popular tests. Therefore, all testing frameworks provide a means to test for equality. JUnit, for example, provides a complete amount of overloaded equality tests. C++ templates can do that as well with less code. For more complex data types, such as strings, it might be hard to see the difference between two values given, when they are simply printed in the error message.

```
template <typename EXPECTED, typename ACTUAL>
std::string diff_values(EXPECTED const &expected
                        ,ACTUAL const & actual){
// construct a simple message...
    std::ostringstream os;
    os << "(" << expected<<","<<actual<<")";
    return os.str();
}
// special overloaded cases for strings
std::string diff_values(
 std::string const &,std::string const &);
std::string diff_values(
 char const * const &exp,std::string const &act);

template <typename EXPECTED, typename ACTUAL>
void assert_equal(EXPECTED const &expected ,
                  ACTUAL const &actual
                  ,char const *msg,
                  char const *file,int line) {
  if (expected == actual) return;
  throw cute_exception(
   msg + diff_values(expected,actual),file,line);
}
#define assertEqualsm(msg,expected,actual) \
    cute::assert_equal(\
     (expected),(actual),msg,__FILE__,__LINE__)
#define assertEquals(expected,actual) \
    assertEqualsm(\
    #expected " expected but was " #actual,\
    expected,actual)
```

Listing 2

```
void anothertest(){
  assertEquals(42,lifeTheUniverseAndEverything);
}
```

One means to implement **assertEquals** would be to just **#define** it to map to **t_assert((expected)==(actual))**. However, from my personal experience of C++ unit testing since 1998, this is too simplistic in cases where the comparison fails. Especially for strings or domain objects seeing the difference between two values is often important for correcting a programming mistake. In my former life, we had custom error messages for a failed string compare to spot the difference easily. Therefore, CUTE provides a template implementation of **assert_equal** that again is called by a macro, to enable file position gathering.

I speculated (maybe wrongly) it would be useful to be able to specify your own mechanism to create the message if two values differ, which also is implemented as a to-be-overloaded template function. (See Listing 2.)

I encourage readers to criticize this design to help me to come up with something simpler.

Listening customization

You've already seen, that the **runner** class template can be specialized by providing a listener. The **runner** class is an inverted application of the Template Method Design Pattern **![GoF]**. Instead of implementing the methods called dynamically in a subclass, you provide a template parameter that acts as a base class to the class **runner**, which holds the Template Methods **runit()** and **operator()**. (See Listing 3).

If you look back to **runner::runit**, you will recognize that if any reasonable exception is thrown, it would be hard to diagnose, what the reason for an error is. Therefore, I included **catch** clauses for **std::exception**, **string** and **char** pointers to get information required for diagnosis. The demangling is required for GNU g++ to get a human-readable information from the exception's class name.

```
} catch (std::exception const &exc){
    Listener::error(t,test::demangle(
     exc.what()).c_str());
} catch (std::string &s){
    Listener::error(t,s.c_str());
} catch (char const *&cs) {
    Listener::error(t,cs);
```

Again I ask you for feedback if doing so seems over-engineered. Are you throwing strings as error indicators?

As you can see, there are a bunch of methods delegated to the base class given as **runner**'s template parameter (**begin**, **end**, **start**, **success**, **failure**, **error**). The default template parameter **null_listener** applies the Null Object Design Pattern and provides the concept all fitting Listener base classes. (Listing 3).

```
struct null_listener{
  // defines Contract of runner parameter
  void begin(suite const &s, char const *info){}
  void end(suite const &s, char const *info){}
  void start(test const &t){}
  void success(test const &t,char const *msg){}
  void failure(test const &t,
               cute_exception const &e){}
  void error(test const &t,char const *what){}
};
```

So whenever you need to collect the test results or you want to have a nice GUI showing progress with the tests, you can create your own specific listener.

Again you can employ an inverted version of a GoF Design Pattern, to stack listeners. This is application of an inverted Decorator using C++ templates, for example to count the number of tests regarding their category, see Listing 4.From the schema in Listing 4, you can derive your own stackable listener classes, e.g. one showing the progress of running the tests and their results in a GUI. If you do so, share your solution.

Test extensions

With the idea of allowing all non-parameter Functors to be eligible as tests, it is relatively simple to provide test-wrappers for different kind of functionality.

Exception testing

Good practice of unit testing is also to check if things go wrong as intended. So you want to expect a specific exception from a test functor. The code to do that can easily be canned for reuse in a template and with CUTE it is activated by a macro call like:

```
  suite s;
  s += CUTE_EXPECT(
   functor_that_throws(),std::exception);
```

```
template <typename Listener=null_listener>
struct runner : Listener{
    runner():Listener(){}
    runner(Listener &s):Listener(s){}
    bool operator()(test const &t){
        return runit(t);
    }
    bool operator()(suite const &s,
                    char const *info=""){
        Listener::begin(s,info);
        bool result=true;
        for(suite::const_iterator it=s.begin();
         it != s.end();++it){
            result = this->runit(*it) && result;
        }
        Listener::end(s,info);
        return result;
    }
private:
    bool runit(test const &t){
        try {
            Listener::start(t);
            t();
            Listener::success(t,"OK");
            return true;
        } catch (cute_exception const &e){
            Listener::failure(t,e);
        } catch(...) {
            Listener::error(
             t,"unknown exception thrown");
        }
        return false;
    }
};
```

Listing 3

```
template <typename Listener=null_listener>
struct counting_listener:Listener{
    counting_listener()
    :Listener()
    ,numberOfTests(0),successfulTests(0)
    ,failedTests(0),errors(0),numberOfSuites(0){}

    counting_listener(Listener const &s)
    :Listener(s)
    ,numberOfTests(0),successfulTests(0)
    ,failedTests(0),errors(0),numberOfSuites(0){}

    void begin(suite const &s, char const *info){
        ++numberOfSuites;
        Listener::begin(s,info);
    }
    void start(test const &t){
        ++numberOfTests;
        Listener::start(t);
    }
    void success(test const &t,char const *msg){
        ++successfulTests;
  Listener::success(t,msg);
    }
    void failure(test const &t,
                 cute_exception const &e){
        ++failedTests;
        Listener::failure(t,e);
    }
    void error(test const &t,char const *what){
        ++errors;
        Listener::error(t,what);
    }
    int numberOfTests;
    int successfulTests;
    int failedTests;
    int errors;
    int numberOfSuites;
};
```

Listing 4

The implementation of that mechanism is as you have expected (see Listing 5).

```
template <typename EXCEPTION>
struct cute_expect{
  test theTest;
  std::string filename;
  int lineno;
  cute_expect(test const &t,char const *file,
              int line)
  :theTest(t), filename(file), lineno(line){}
  void operator()(){
    try{
      theTest();
      throw cute_exception(
       what(),filename.c_str(),lineno);
    } catch(EXCEPTION &e) {
    }
  }
  std::string what() const{
    return theTest.name() + " expecting "
           +
test::demangle(typeid(EXCEPTION).name());
  }
};
#define CUTE_EXPECT(tt,exc) \
    cute::test(cute::cute_expect<exc>(\
     tt,__FILE__,__LINE__),tt.name())
```

Listing 5

No need to implement the try-catch again by hand for testing error conditions. What is missing, is ability to expect a runtime error recognized by the operating system such as an invalid memory access. Those are usually signalled instead of thrown as a nice C++ exception.

With a similar wrapper you can implement a class for repeatedly running a test. There isn't even a template required.

Member functions as tests

Having `boost::bind()` at your disposal, it is easy to construct a functor object from a class and its member function. Again this is canned in a macro that can be used like:

```
CUTE_MEMFUN(testobject,TestClass,test1);
CUTE_SMEMFUN(TestClass,test2);
CUTE_CONTEXT_MEMFUN(contextobject,TestClass,test3);
```

The first version uses object **`testobject`**, an instance of **`TestClass`**, as the target for the member function **`test1`**. The second version creates a new instance of **`TestClass`** to then call its member function **`test2`** when the test is executed. The last macro provides a means to pass an additional object, to **`TestClass`**' constructor, when it is incarnated. The idea of incarnating the test object and thus have its constructor and destructor run as part of the test comes from Kevlin Henney and is implemented in Paul Grenyer's testing framework Aeryn.

The macro **`CUTE_MEMFUN`** delegates its work to a template function as shown in Listing 6.

The template function **`makeMemberFunctionTest`** employs **`boost::bind`** to create a functor object that will call the member function fun on object **`t`**, when called. Again we can employ C++ reflection using **`typeid`** to derive part of the test object's name. We need to derive the member function name again using the preprocessor with a macro. To allow to use also **`const`** member functions, the template function comes in two incarnations, one using a reference as shown and the other one using a **`const`** reference for the testing object.

Test object incarnation

I will spare you all details, but give you the mechanism of object incarnation and then calling a member function for the case, where you can supply a context object. (See Listing 7).

This will allow you to use test classes with a constructor setting up a test fixture and a destructor clearing it again. So there is no longer a need for writing Java like **`setUp()`** and **`tearDown()`** methods.

Limitations and outlook

One big difference between C++ and other languages is the lack of method-level introspection. The only means for getting a list of tests to execute is having a programmer specifying it, i.e., by registering test objects somewhere. If anybody is aware of how to get rid of that and have tests registered automatically please let me know. CppUnit compensates this by the ability to automatically load shared libraries with test classes. On the other hand, this makes writing tests, using CppUnit and its implementation more complex.

CUTE is still in a nascent state at the time of this writing. It comes with a small test suite for itself, but especially with all the templates it might still suffer from problems in its use, not yet encountered by me. If you haven't yet written unit tests for your code, try starting now using CUTE and tell me how it feels and works. You can download the currently released version of CUTE in source form from my wiki web at http://wiki.hsr.ch/PeterSommerlad/ .

There are many ideas for extending CUTE to make it a more convenient environment to live in. For example, better IDE integration to directly navigate from a failed test is a must for professional use. Tell me your ideas, or just implement them. Thank you in advance. ■

References

[Aeryn] Paul Grenyer http://www.aeryn.co.uk

[CppUnit] http://cppunit.sourceforge.net/cppunit-wiki

[CppUnitCookbook] http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html

[GoF] Gang of Four, E. Gamma, R. Helm, R. Johnson, J. Vlissided: *Design Patterns - Elements of Reusable Object-Oriented Design*

[JUTLAND] Kevlin Henney, Java Unit Testing Light Adaptable 'N' Discreet, presentation at JAOO 2005 and private communication

```
template <typename TestClass>
test makeMemberFunctionTest(TestClass &t,
     void (TestClass::*fun)(),char const *name){
  return test(boost::bind(fun,boost::ref(t)),
     test::demangle(typeid(TestClass).name())+
     "::"+name);
}
#define CUTE_MEMFUN(testobject,TestClass,\
        MemberFunctionName)\
    cute::makeMemberFunctionTest(testobject,\
        &TestClass::MemberFunctionName,\
         #MemberFunctionName)
```

Listing 6

```
template <typename TestClass,typename MemFun,
 typename Context>
struct
 incarnate_for_member_function_with_context_object
 {
  MemFun memfun;
  Context context;

incarnate_for_member_function_with_context_object(
   MemFun f,Context c)
  :memfun(f),context(c){}
  void operator()(){
    TestClass t(context);
    (t.*memfun)();
  }
};
template <typename TestClass, typename MemFun,
          typename Context>
test makeMemberFunctionTestWithContext(
 Context c,MemFun fun,char const *name){
  return test(

incarnate_for_member_function_with_context_object
   <TestClass,MemFun,Context>(fun,c),
   test::demangle(typeid(TestClass).name())
   +"::"+name);
}
```

Listing 7

From CVS to Subversion

Thomas Guest reflects on migrating his organisation's version control system from CVS to Subversion.

Introduction

The time had come to upgrade our source control system. As CVS users, the obvious choice was Subversion. This article describes how the upgrade went and provides some practical advice for anyone considering making a similar move.

The reason for change

CVS is an excellent source control system: fast, powerful and flexible. We had no concerns regarding its reliability and some effort had been put into integrating it into our automated build, test and release system. What's more, everyone in the team knew how to use CVS and how to work around its wrinkles. We all had our favourite clients. Why ever would we want to change?

There were a number of reasons:

- The team had grown and so had the codebase. The CVS server no longer served high volumes of files as quickly as we'd have liked.
- As the codebase grew, it had become apparent that some files were in the wrong places or had the wrong names. CVS supports versioning of files but not of file-systems, meaning that we couldn't fix these issues in a controlled way. Subversion fixes this CVS limitation.
- CVS does not support atomic commits (see Sidebar) – another feature built in to Subversion.
- Subversion sets out to be a 'compelling replacement for CVS' and, after a quick skim through the documentation, it looked as though the transition would be painless.

Evaluation

We did pause – albeit briefly – to consider whether an alternative version control system might better meet our needs. We couldn't think of any. The decision was somewhat political since we'd recently been acquired and the parent company had its own preferred version control system. The move to Subversion could be passed off as an upgrade of our current system rather than a truly subversive act.

Our next step, then, was to evaluate Subversion. The aims of the evaluation were:

- measure the performance of Subversion
- build some expertise in Subversion administration
- confirm Subversion's core capabilities consider how best to actually use Subversion
- if all looked good, put a transition plan in place.

Thomas Guest is an enthusiastic and experienced computer programmer. He has developed software for everything from embedded devices to clustered servers. His website can be found at http://www.wordaligned.org and you can contact him at thomas.guest@gmail.com

Atomic Commits

Suppose a single logical change to the codebase – a bugfix perhaps – requires six files to be changed. The programmer makes the change and commits the new versions of the files to the CVS repository. Although a single commit command is executed, as far as CVS is concerned six changes have been made, and each individual file moves to its own new revision. If another programmer wishes to patch the bugfix to another code branch, all six files will need to be patched – but it's tricky to find this out from CVS. Information has been lost.

Subversion solves this problem with a simplified change model: version numbers apply to the repository as a whole, and each

Clearly, the first step was to set up a Subversion server and import a snapshot of our CVS repository to practise on.

Setting up the trial server

Setting up the trial server was straightforward. On (Mandriva) Linux, after the usual package selection and update process we had an `svn` user ready to serve the repository, and an `/etc/xinet.d/svnserve` configuration file, the contents of which are shown in Listing 1.

Most of the contents of this configuration file should be easy to figure out. The actual program which will serve the repository is **`/user/bin/svnserve`** (run as a daemon by **`xinetd`**) and it should be run by user `svn` with arguments **`-i`** (inetd mode) and **`-r /var/lib/svn/repositories`** (root of directory to serve).

Once we had created the repository (see next section) in the configured location, we enabled the Subversion server as follows:

```
su                          # root runs xinetd
chkconfig svnserve on       # enable svnserve service
xinetd restart
```

```
# default: off
# description: svnserve is the server part
# of Subversion.
service svnserve
{
  disable      = yes
  port         = 3690
  socket_type  = stream
  protocol     = tcp
  wait         = no
  user         = svn
  server       = /usr/bin/svnserve
  server_args  = -i -r /var/lib/svn/repositories
}
```

Figure 1

What we found, then, on the performance side, was that the routine management of a working copy was much quicker

Note here that although the root user starts the **xinetd** service, the svn user actually owns and serves the repository.

Server options

There are two main options for serving a Subversion repository:

- using the custom **svnserve** server
- using Apache **httpd** with **mod_dav_svn**

A full discussion of the pros and cons of these options can be found in the Subversion book: [Subversion].

This discussion is summarised in a table [Subversion2].

We opted to use the custom **svnserve** server because, according to the documentation it would be easier to set up and somewhat faster.

Whilst I have no experience of using Apache as a Subversion server, I can certainly confirm that **svnserve** is simple to set up.

Importing a copy of the CVS repository

Creating the trial repository was a little more time consuming. To perform realistic tests we needed something like a full import of our CVS repository.

Subversion provides a Python program, **cvs2svn**, to perform this import. One thing you really shouldn't do is try to import a live CVS repository, which is of course a moving target. One thing we equally didn't want to do was take down the CVS server for any period of time. Fortunately, we kept a mirror of the live repository; by taking a copy of this mirror (when it wasn't being mirrored!), we gave ourselves something to import.

The documentation for **cvs2svn** is rather thin. In fact, at the time of writing this article, there's not a huge amount over and above what the command line tells you:

```
cvs2svn --help
```

Fortunately, the program works. Most of the options aren't even required if you're happy to go with the default repository layout, default database backend, default keyword expansion mode, default end-of-line style and so on. And, if **cvs2svn** does hit problems – which will almost certainly be caused by 'Garbage In' – it exits smartly and tells you what to do next.

In our case, we had to clear up a little tag ambiguity, and then we were off. The import took about four hours – this is for a CVS repository consisting of about 64000 files, a few hundred tags and branches, and occupying around 12Gb on disk.

Choice of database layer

At revision 1.3, Subversion provides a couple of database backends:

- a Berkeley DB database
- FSFS, where data is stored in ordinary flat files, using a custom format.

The pros and cons of the two choices are discussed in the Subversion book [Subversion3] and summarised in a table [Subversion4].

Although the FSFS backend is less mature it looked more suitable in every other respect. The Subversion repository create tool, **svnadmin create**, treats FSFS as the default database backend – and so does **cvs2svn**. We decided to go with this default and have had no complaints.

More evaluation

To our surprise and disappointment, the speed of clean checkouts (by 'clean' I mean checking out the entire codebase into a new directory, rather than simply updating an existing working copy) was underwhelming. CVS sets a hard act to follow here since one of its strengths is its speed, but I simply couldn't imagine Subversion claiming to be a compelling replacement for CVS unless it was equally fast. In fact, head-to-head, on the same platform, our tests showed CVS to be measurably quicker for clean checkouts.

What the trials did indicate was that disk access rather than network bandwidth was the main source of pain. Every time it checks out a file, Subversion replicates the base version of that file and its properties ('Properties' is the Subversion term for metadata associated with a file – such as whether it's executable, for example.) into a hidden .svn directory, so for every 100 files you checkout, at least 500 files will be created on disk.

This replication is quite deliberate and is based on the principle that disk-space is cheaper than network bandwidth. Subversion makes full use of the cached file copies in your working area – so, for example, common operations such as viewing your modifications to a file, or reverting these modifications, do not require any interaction with the server.

What we found, then, on the performance side, was that the routine management of a working copy was much quicker. Clean checkouts took time, yes, but use of the **svn update** command keeps these to a minimum. In fact, the only user who frequently performed clean checkouts was our overnight automatic build.

Everything else went very well. Clearly, the authors of Subversion had done a great job in fixing the problems with CVS, and they'd done so – at least from a user's perspective – by simplifying it.

The transition plan

There never seems to be a good time to change tools. There will always be releases to make, builds to test, critical patches to issue, and it's understandably hard to justify even the smallest amount of downtime in such real, customer-facing activity. Indeed, when there's lots of this real work to be done it's equally hard to dedicate the time to take proper care when executing such a tool change.

On the other hand, by taking such an argument to the extreme, software developers end up stuck using Visual Studio 6.0 and grumbling (unfairly) about Microsoft's poor support for C++.

We were, then, keen to proceed. Our transition plan was simple. The timescale was short but manageable – and we knew that if we missed the slot, we wouldn't get another chance for a while.

The problem we had was with binary files which had (wrongly) been checked into CVS as text files

As part of the evaluation, we'd created some backup scripts to mirror a Subversion repository to our backup machine. Next, we set up a migration script to:

- disable scheduled jobs which might get in the way
- take down the CVS server
- copy the CVS repository
- import this copy using **`cvs2svn`**
- log any problems occurring in any of these steps.

So, to initiate the transition, all we had to do was schedule the migration script to run overnight. In the morning, if the log files were clean, we could kick off the Subversion backups, point the Subversion server at the newly imported repository and start it up, restart the CVS server in readonly mode, and we'd be done.

What we failed to do

A number of items on the evaluation and transition plan never happened. We didn't create any local training materials – it didn't seem necessary, given the high quality of Subversion's built in documentation, and the fact that we all knew CVS (see Sidebar: Subversion for CVS users). We ordered printed copies of *Version Control with Subversion* and *Pragmatic Version Control Using Subversion*, set up an FAQ page on the Wiki that did little more than collect together a few links to offsite URLs, and left it at that.

Despite encouragement, no-one had bothered to use the trial repository as a sandbox for experimenting with Subversion (apart from the individual actually running the trial). So, the evaluation of the product's usability and basic functionality was down to just one person. Again, this turned out not to be a problem – and we weren't really being lax when you consider how many open source projects have switched, or are switching, to Subversion. We just _knew_ Subversion worked.

We quite deliberately didn't plan any reorganisation or pruning of the CVS repository before importing it: Subversion would allow us to make such changes in a better controlled way, once we got to the other side. For similar reasons, we didn't change keyword expansion properties on import. Again, Subversion allows you to manage such properties better than CVS does, and now was not the time to start arguing whether or not we really thought keyword expansion was a good idea (keyword expansion is discussed and assessed below - Alan).

We didn't fix any of our build scripts in advance. As part of the evaluation we'd grepped the source for all such scripts and it turned out you could count the number of scripted calls to **`cvs`** on the fingers of one hand. We were confident we could fix these pretty much as soon as our Subversion server went live.

We didn't even bother evaluating any advanced Subversion clients. I used the command line almost exclusively for experimentation: others were happy to defer setting up TortoiseSVN, Subclipse, psvn, etc., until they actually had to.

The one crucial item we neglected from our plan was to perform acceptance tests on the freshly imported repository. Fortunately we discovered the problem with our carelessness almost immediately and were able to recover swiftly.

The problem

The problem we had was with binary files which had (wrongly) been checked into CVS as text files. On import, by default, **`cvs2svn`** does a couple of things to text files which can seriously damage binary files:

- keyword-expansion is enabled meaning that byte sequences which match patterns such as **`$Id: $`** get changed when you check the file out.

  (Strictly speaking, **`cvs2svn`** sets **`svn:keywords`** on CVS files to **`author id date`** if the mode of the RCS file in question is either **`kv`**, **`kvl`** or not **`kb`**.)
- the end-of-line style property is set to **`native`**, meaning again that the binary file you check out may not be the one you checked in, since Subversion makes sure end-of-line sequences are the ones preferred by your client platform.

Subversion for CVS Users

If you're familiar with CVS then Subversion will also seem familiar. This is hardly surprising since Subversion's stated aim is to be a compelling replacement for CVS. So, the terminology is almost identical: you 'check out' files from a 'repository', you edit them, you 'diff' files to see what you – and others – have changed, and you 'check in' your changes. You 'update' your working copy to merge in changes made by other team members. You can 'log' what changes have been made. You can 'branch' a project by copying it from the 'trunk' into a new place in the repository; similarly you can 'tag' a fixed version of a project by copying it into a new place in the repository. You can 'merge' changes made on the branch back to the trunk, and vice versa.

If you use the command line client, **`svn`**, the command line arguments are often identical to the ones used with the **`cvs`** client (**`svn commit`**, **`svn checkout`**, **`svn status`**, **`svn annotate svn diff`**, **`svn log`**, etc.).

The areas where CVS and Subversion differ are generally where Subversion fixes a CVS deficiency or where Subversion actually manages to simplify things. For example, as already mentioned, Subversion fixes a well-known CVS deficiency by allowing you to move files and directories; and Subversion's transactional model means that a version number (a revision number, in Subversion terminology) is an incrementing integer applied to the repository as a whole, which is easier to work with than the dot-separated version numbers which apply to each CVS controlled file.

More information for CVS users migrating to Subversion can be found at: http://svnbook.red-bean.com/en/1.2/svn.forcvs.html

We'd messed up but fortunately we'd messed up in an immediately obvious way: a number of binaries were broken, to the point that they wouldn't even execute

We'd messed up but fortunately we'd messed up in an immediately obvious way: a number of binaries were broken, to the point that they wouldn't even execute.

This is one of those mistakes you only make once (until you make it the next time and kick yourself even harder, that is). I guess we were lulled into a false sense of security: everything seemed to be working so smoothly ... Subversion is better than CVS at handling binary files ... everything had been working fine with CVS, so our CVS repository must be fine ... **`cvs2svn`** would spot any problems.

Of course, our CVS repository wasn't fine. We'd got away with binary files marked as text for the simple reason that most of these files had been used on Linux only.

Acceptance tests

What makes this mistake so chastening is the fact that a basic acceptance test of the new repository would have been both simple and scriptable:

```
#!/bin/sh
  cvs co CVSARCHIVE fromcvs
  # Checkout from CVS, on the trunk
  svn co SVNREPOS/trunk fromsvn
  # Checkout from SVN, on the trunk
  diff -q -r fromcvs fromsvn > all_diffs
  # Spot the difference
```

If the `all_diffs` file is empty, the CVS and Subversion checkouts are byte-for-byte compatible.

Unfortunately the `all_diffs` file wasn't empty. Remember those keyword expansions? Subversion is clever enough to replace CVS version numbers with its own revision numbers and as a result the files differ when checked out. Keyword expansion really is a bad idea!

Similarly, a number of text files were different because Subversion had tidied up inconsistent line endings.

So, there were plenty of false hits as well as a list of files we needed to run **`cvsadmin -kb`** on.

Incidentally, we could have chosen to clean up the files during import by passing some more parameters to **`cvs2svn`**: a suitable combination of **`--mime-types=FILE`**, **`--eol-from-mime-type`** and **`--no-default-eol`** options would have done the job. We decided, though, that the proper solution was to fix the root cause of the problem.

Recovery

So, we had to delay by a day to reinstate CVS, run the text-to-binary corrections, re-run the migration, perform acceptance tests. This time we were more cautious and we also tested builds made from the clean Subversion checkout.

Scheduled backups

I won't go into detail here about the differences between CVS and Subversion. There's plenty of solid documentation already available.

One thing worth mentioning is the strategy we adopted for Subversion backups. Previously, our CVS repository had been mirrored to a backup machine using a **`rsync`** job scheduled to run every couple of hours. Tape backups of this mirror were kept offsite.

I had some reservations about this strategy, particularly since (thanks to our hyperactive and insomniac automatic build user) the CVS archive was rarely quiet. Simply treating the CVS archive as a bunch of files – which is what **`rsync`** does – seemed risky. Would the mirror be in good shape if **`rsync`** ran in parallel with a check-in?

Subversion provides the ability to make a hot backup of a live repository using the **`svnadmin hotcopy`** command. The repository dump can be loaded into a Subversion repository using **`svnadmin load`**. So, something as simple as:

```
svnadmin hotcopy /path/to/live/repository
  /path/to/mirror/repository
```

creates a full mirror of the live repository – if you're prepared to wait a while, that is.

Once this mirror has been created, it can be maintained by merging in incremental changes using **`svnadnim dump --incremental`** to dump the changes and **`svnadmin load`** to load them into the repository mirror.

How much should you import?

We never really explored the idea of restricting what we imported into Subversion. **`Cvs2svn`** offers lots of choices:

- you can topskim your repository, meaning you get no history, no branches – a fresh start.
- you can import absolutely everything, meaning you get every single branch ever made, every bungled thirdparty import – everything!
- you can import anything in between these two extremes by selecting which tags and branches to import.

It's hard to argue against the 'import everything' option: source control is all about tracking and managing changes, so why should any change ever be thrown away? And, as already mentioned, Subversion does provide a 'delete' option (of course, anything you delete can be recovered), so you can tidy up at any point.

Since the initial import we've exercised **`svn delete`** rather a lot, and every time we need to upgrade a vendor branch we end up moving things. I still wonder if something closer to a topskim wouldn't have been better. We'll never know. And I secretly wish we'd accidentally-on-purpose turned off keyword expansion!

All too often a software upgrade means buying in to more features and more complexity

Changing tools revisited

As already mentioned, tool changes can be hard to justify. However, despite the hiccup in the migration, CVS to Subversion required little effort and led to almost no downtime. Perhaps the stated reasons for change didn't seem that compelling – if we'd lived without atomic commits and version controlled file systems for so long, surely we didn't really need them? The paradox here is that you can't really appreciate how important these features are until you actually use them – and so, from the other side of the change, we wonder how we ever did without them!

What I like most about Subversion though is that, from both a user's and an administrator's perspective, it's simpler than CVS. All too often a software upgrade means buying in to more features and more complexity. Think of all those new people joining the team, some of whom may never have used source control. Consider explaining the Subversion model for repository revisions, branches, tags. Now consider explaining the same topics using the CVS model. Clearly less time will be needed getting people up to speed.

Conclusions

Everyone likes the new source control system, which is important – freedom of choice may be acceptable for editors, web browsers, and even operating systems, but a team really must agree to share a source control system. It's important to like the tools you use every day.

CVS was good but Subversion is better. As already mentioned, head-to-head, on the same hardware, CVS managed to beat Subversion on clean checkouts – but who said we had to use the same hardware? We invested in a powerful new computer to serve our powerful new source control system, so even clean checkouts are quicker. Routine operations on a working copy are much quicker.

Upgrading build scripts did indeed turn out to be simple.

Four clients are in active use (five if you count the command line client, **`svn`**, itself). I use the **`psvn`** Emacs integration, which is very similar to **`pcl-cvs`**. Subclipse, TortoiseSVN and kdesvn are also popular with Eclipse, Windows and KDE users respectively.

So, CVS to Subversion makes good sense, but do beware of pitfalls in the import procedure. ■

Further reading

More Subversion tips can be found at: http://blog.wordaligned.org/articles/category/subversion

Other sources

- CVS: http://www.nongnu.org/cvs/
- Subversion: http://subversion.tigris.org/cvs2svn
- A Python script that converts a CVS repository to a Subversion repository: http://cvs2svn.tigris.org/
- Subclipse, A Subversion Eclipse plugin: http://subclipse.tigris.org/
- TortoiseSVN, A Subversion client implemented as a windows shell
- extension: http://tortoisesvn.tigris.org/
- psvn, Subversion interface for emacs: http://svn.collab.net/repos/svn/trunk/contrib/client-side/psvn/psvn.el

Credits

My thanks to the editorial team at Overload for their help with this article.

References

[Subversion]
http://svnbook.red-bean.com/en/1.2/svn.serverconfig.html

[Subversion2]
http://svnbook.red-bean.com/en/1.2/svn.serverconfig.html#svn.serverconfig.overview.tbl-1

[Subversion3]
http://svnbook.red-bean.com/en/1.2/svn.reposadmin.html#svn.reposadmin.basics.backends

[Subversion4]
http://svnbook.red-bean.com/en/1.2/svn.reposadmin.html#svn.reposadmin.basics.backends.tbl-1