# contents

Letters to the Editor 6

From Waterfall to EVO in a Medium Size Norwegian Software House
Trond Johansen 10

Multiple Streams Going Nowhere
Paul Grenyer 12

STABLE INTERMEDIATE FORMS: A Foundation Pattern for Derisking the Process of Change
Kevlin Henney 16

C Abuse
Thaddaeus Frogley 22

A Pair Programming Experience
Randall W Jensen 22

The Developer's New Work
Allan Kelly 25

C++ Properties - A Library Solution
Lois Goldthwaite 28

## credits & contacts

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

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

**Advisors:**

**Phil Bass**
phil@stoneymanor.demon.co.uk

**Thaddaeus Frogley**
t.frogley@ntlworld.com

**Richard Blundell**
richard.blundell@metapraxis.com

**Advertising:**
**Chris Lowe**
ads@accu.org

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

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

**Information and Membership:**
Join on the website or contact

**David Hodge**
membership@accu.org

**Publications Officer:**
**John Merrells**
publications@accu.org

**ACCU Chair:**
**Ewan Milne**
chair@accu.org

# Editorial: "They" Have Their Reasons

People, by and large, work by trying to achieve goals – and it is by understanding their goals that we can best understand their behaviour. That is why "user stories" are such an effective way of capturing requirements (most approaches to requirements capture are effective when they are used with a focus on what is being attempted). But, as anyone that has done requirements capture should be able to tell you, people tend to be poor at explaining what their goals are. Without guidance they will focus on how they expect these goals to be achieved.

Contrast the following directions to a colleague's desk:

> *"Go through those double doors, across the office and through the next ones, down the stairs to the next floor, turn left and go through the security door, follow the corridor around to the left and right, when you go through the next door he's over to the right by the window."*

With:

> *"About twenty feet in that direction and one floor down."*

Which is easier to understand? Or easier to implement? I'd take the goal oriented version – which tells me where I'm trying to get to – every time.

More importantly, the first explanation is much more susceptible to changes in circumstances: the stairs being out of use, extra doors being introduced on the corridor...

A couple of years ago I spent several months working with business analysts who regularly produced requirements specification documents that read like the first quote above. Actually, they were worse: although the business analysts avowed no special knowledge of computers, and especially of user interface design, they included screen layouts. I was involved for several reasons, but two important ones were that the customer didn't accept the resulting software (it didn't address their requirements) and that the requirements capture process was far too slow (at the rate it was going it would take several times the agreed time-scale for the project).

It didn't take long to establish that the business analysts didn't enjoy writing this stuff. Or that the customers struggled to approve it (or "accepted it" without agreeing for contractual purposes). Or that errors and omissions were not detected until late in the development cycle (integration testing or acceptance testing). Or that the developers were frustrated into blindly implementing things they didn't pretend to understand. And if a change in understanding required changes to the product it was an intractable problem to find all the documents affected.

Fortunately, by the time I got involved, the project was suffering sufficient pain that enough people were willing to try something else (so long as I took the blame if it didn't work). Having quickly read Cockburn's *"Writing Effective Use Cases"* I chose to introduce goal oriented "stories" describing what people would be doing with the system we were developing. We also dispensed with screen layouts and substituted lists of items to be input and presented. The customers found the resulting documents more accessible and contributed more to their creation, the business analysts found the documents easier to produce, and the developers felt they could identify and deliver what was wanted. Everyone thought it an improvement.

Why then had the "old way" become established? Asking the business analysts got responses along the lines of "we don't like it, but that is what they [the developers] want". The developers had a different version "we don't like it, but they [the business analysts] have to do it that way for customer sign off". Somehow, no one had been happy, but had just accepted that things were the way they were because "they" needed it that way.

"They" is one of those stereotypes of social life – a faceless other that behaves in inexplicable (and often damaging) ways. Users try to do the weirdest things with the software we supply them, managers seem determined to stop the work getting done, prospective employers eliminate talented individuals during the recruitment process, developers show no interest in avoiding problems, accountants shut down successful projects. "They" cause many of the problems and irritations we face in life. "They" are stupid, malicious or ignorant.

Nonsense! If you can find and talk to them you will find that "they" are normal human beings trying to achieve reasonable goals in reasonable ways. And, all too frequently, "they" are just as dissatisfied with the state of affairs as you are.

- When I'm using a piece of software I don't suddenly lose all sense – maybe it is hard to figure out how to achieve my objectives. I'll try things that make sense to me to try – which is not always what the developer expected. (Even when the developer has been diligent about getting feedback on the user interface design.)
- If I'm running a project then I don't forget that code needs to be written, but sometimes ensuring that the functionality meets the need or ensuring that funding continues requires something else gets done first.
- If I'm recruiting I need to avoid people that won't be effective in the organisation – bringing someone disruptive into a team costs their time and that of others. Given that cost is it surprising that employers are not prepared to "take a chance" when there is anything that raises doubts about the suitability of a candidate.
- If I'm developing software I can only tackle so many issues at once. If an organisation lacks a repeatable build process and version control then these are things that need fixing before looking at the proposed list of new features. Some problems are not serious enough to warrant effort right now.

- If I'm funding work I want to see a return (not necessarily financial) that is better than alternative uses of those funds. The way in which software developers sometimes report results can make it very hard to assess that return.

I don't consider any of these goals inexplicable or unreasonable – nor should you.

It is a refusal to consider the reasons for the way "they" act that builds the problems, and labelling them "they" is an abdication of rationality. While there is a role for "they" and "we" in thinking it is one that defines allegiances and trust, not one that helps to resolve problems.

Some of this thinking seems to influence the content of Overload: many potential authors think that "they" (the editorial team) are only interested in C++, while the editorial team wonder why "they" (the authors) hardly ever submit material relating to other languages. Admittedly we do get a few Java articles, but where are the C, Python, C# and SmallTalk articles? I know there are members that are interested in these technologies, so there should be both an audience and people with the knowledge to write something. Come to think of it, if you think "they" are not publishing the right articles why not get involved? You could write articles, you could even join the team – we've not recruited any "new blood" to the Overload team for two years now. Maybe "they" could include you?

## ACCU Certification

As I'm writing this a discussion has sprung up on `accu-general` about a topic that resurfaces every year or so: "why is there no effective qualification for software developers?" There are organisations (like the BCS, IEE or EC[UK]) that might be thought suitable for supporting such – but "they" don't provide anything the list members feel is appropriate. This same issue was raised in Neil Martin's keynote at the last ACCU conference when he suggested that the ACCU step in and address this need. As a result, the ACCU Chair (Ewan Milne) asked me to arrange a "Birds of a Feather" session for those interested in exploring this possibility, and to represent the committee there.

The session was well attended, and there seemed to be a strong consensus that there were potential benefits for both developers and employers in some sort of certification-of-competence scheme. It was also thought that it would be a good idea for ACCU to get involved in producing such a scheme. Questions were raised about what was involved in becoming a certificating body, what it was practical to certify and what the mechanism for certification might be. There seemed to be a lot of interest – so Neil promised to research the certification issues and I took email addresses for those interested in participating in further discussion and got the `accu-certification` mailing list set up.

Clearly there was a misunderstanding: I expected "they" (the people that signed up) would involve themselves in doing something. It seems that those that signed up expected that a different "they" (Neil, myself or the committee) would do something. In practice only Neil did something – he reported back as promised: ACCU could reasonably easily get itself recognised as a "certification body" for this purpose. The details of what is involved were circulated via the mailing list. And that was the end of it until the discussion on `accu-general`.

It is easy to be critical and say that "they" should do something. In this case that "they" (the ACCU) should do something about certifying developers as being competent. But just think for a moment: you are a member of ACCU, and ACCU works through members volunteering to do things. So you are one of this particular "they", and you know exactly why "they" are doing nothing – because you are doing it yourself.

In practice though, I feel that the ACCU already does provide a useful qualification: I did hundreds of "technical assessments" for a client last year – most candidates failed in ways that gives cause for concern about an industry that employs them. (Interestingly I had feedback both from the group that I was working with about how competently the "passes" fitted in and also from other groups in the client organisation that decided to employ some of those I had failed at interview – and then found them deficient.) The qualification that ACCU provides? I can't recall any candidates that mentioned the ACCU on their CV failing the technical part of the process (while the client wasn't prepared to exempt them from the assessment on that basis, the manager selecting the candidates to interview noticed this).

## Before you go

I was very pleased with the feedback on `accu-general` and elsewhere to the report by Asproni, Fedotov and Fernandez on introducing agile methods to their organisation (I'd spent some time persuading them to write this material up). As a result I reviewed some material that Tom Gilb had passed me at an Extreme Tuesday Club meeting last year looking for things that might interest Overload readers. Amongst this material was a couple of articles (Jensen and Johansen) that appear in this issue by kind permission of their authors. I hope that these too meet with your approval.

*Alan Griffiths*
`overload@accu.org`

## Copy Deadlines

All articles intended for publication in *Overload 66* should be submitted to the editor by March 1st 2005, and for *Overload 67* by May 1st 2005.

## Copyrights and Trade marks

*Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trademark and its owner.*

*By default the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.*

*Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from Overload without written permission of the copyright holder.*

# Letters to the Editor

## Encapsulate Context

Mr. Griffiths,

When I read the title and abstract for the pattern, I thought it might be really useful. Instead, this Pattern is simply a global wrapped in a shroud. I would like to say, at the outset, that I do believe there is a place for global variables (`std::cout` being an excellent example). I also like the concept of contextually global variables – variables that are global within a given context. To make my argument I offer the following:

A There are two ways a variable becomes global within a program: 1) it is intentionally declared global, or 2) it is passed to every function as a parameter.

B The ENCAPSULATE CONTEXT pattern suggests that there is a single variable that should be passed to all functions within a given context, effectively making the variable global (via A.).

C The ENCAPSULATE CONTEXT pattern as written, does not build a context into the variable, but rather provides a global that can be provided within context. There is nothing to define or constrain the context under which variables within the structure should be accessed.

In short, the functionality provided by the ENCAPSULATE CONTEXT pattern as written can be easily achieved by declaring some global variables in a namespace that is shared by the functions that need access to these variables. This "solution" is far cheaper, more effective, and clearly delineates the level of coupling the functions share, whereas the ENCAPSULATE CONTEXT solution as written only serves to obfuscate the level of coupling, not mitigate it.

Were I a sufficiently adept programmer, I would propose an alternative to the ENCAPSULATE CONTEXT solution as written. Sadly, my skills do not lie in this area and it would take me ages to accomplish this task. At best I can offer some design suggestions.

It strikes me that the container which "collects data together" needs to do more than simply provide a pointer that can be passed around on the parameter line. It should have a built in hierarchy of contexts that allow functions to pass not simply the container, but also information about the context under which the data should be accessed. By providing the context (albeit dynamically) it is possible to limit the scope of the variables that are accessible as the context tree is traversed.

Taking the straw man stock exchange trading system that Allan Kelly used, I would see the calls to the functions something like this:

```
ProcessMarketTrade(msg,
               context->constrain(data_store
                                  || log));
MarketStore::Sell(msg,
                  context->constrain(log));
```

the `constrain` function from within `context` should look something like:

```
MarketContext * MarketContext::constrain(
                        ContextLevel level);
```

In this case, the parameters to `context.constrain()` define the highest point in the hierarchies of data stored within the context to which the function should have access and the call returns a pointer to a variable which has been so constrained (I can envision several objections to this particular implementation, but I hope that the idea is clear). The hierarchy within the `context` container might look something like:

```
config_data
  parms
app_data
  data_store
run_time_data
  log
    error_log
    transaction_log
```

Within the hierarchy there might be variables associated with that point in the hierarchy ... in run_time_data/log/error_log there might be a lock that is used to lock the file while a set of error messages is being written, the file pointer, etc. It is worth noting that you shouldn't be able to pass a context above your current context. Thus, if you had received the container with the context level of `log` you would be able to pass on `log`, `error_log`, or `transaction_log`, but not `parms` or `run_time_data`. Maybe all of this could be done with templates somehow.

This would encapsulate the context and refine how generally global variables might be accessed. A function with access to "log" might not have access to "parms" and this would help with decoupling. It strikes me that this is still a long-winded and complicated way to achieve the same thing as globals within a namespace. Given your concerns, however, that it might be good to be able to recover from the state in which you find the application rather than simply rewriting it so that such coupling is either not needed or explicit, this may be a reasonable approach. Again, I wish I had the necessary skills to adequately program such a structure, but my "back of the envelope" efforts have not come to fruition and I'm wondering if I have grasped the whole of the problem or just a piece.

As a final note, I would like to say that the article was clear, well written, and was on topic for Overload. I do feel that the solution given represents a "bad programming practice" but I readily concede to those more learned in this area then myself. I also think that the solution given is a very complicated way to ignore namespaces for no particular gain and some loss.

Sincerely,

*William Fishburne*

## Allan's Reply

Dear Editor,

I'm rather surprised by the amount of attention my little pattern, ENCAPSULATE CONTEXT, in Overload 63 has generated. However, I regard this as a good thing.

I'm a little disappointed that I was not given the opportunity to respond to Phil Bass's points in the same issue of Overload as his letter appeared. Phil's technical points are all valid, the problem is: how do we balance all these forces? That is the problem that ENCAPSULATE CONTEXT addresses.

Phil is also concerned about the resulting coupling. I too am concerned about this and would draw his attention to the Solution and Consequences sections. These deal with some of the problems which can arise when this pattern is used – or misused. This paper does not claim to be a solution to every programming problem. Like any other pattern this one may be useful sometimes and not others. If a system can be partitioned to avoid this problem then great, if not, then this pattern has a use. Unlike many other patterns this one knows its limitations and highlights these to the reader.

Readers must decide whether this pattern stands or falls. However I am more concerned about a non-technical point Phil makes in his letter. He states "mention of Kevlin Henney, Frank Buschmann and EuroPLoP gives the article an unwarranted air of authority." Let me say that these people are mentioned not to aggrandise the pattern or myself but to acknowledge their assistance in creating the pattern. I am most grateful to these people – and the others mentioned – for all their help. It is in the culture of the patterns community that such assistance is acknowledged.

In no way did I hide anything from these people, there was no attempt to pull the wool over their eyes and slip a dodgy pattern past them. The final choice of words may have been mine but this pattern wouldn't be what it is without the review and comments of others.

These names were not mentioned to create a false authority for the pattern. If the pattern has authority it comes not from mention of these names but form the fact that others have reviewed it on multiple occasions. (In fact, I find it hard to think of any other piece of writing in Overload that has been reviewed as much as this – and that was before the Overload editorial board got to see it.)

Moving onto the comments from William Fishburne. First, let me encourage William to write up his thoughts. I'm sure his skills are up to the job.

Reading William's comments I take two main points. Firstly his suggestion to use a namespace as part of the solution. What he describes sounds like the MONOSTATE pattern – otherwise known as THE BORG. This pattern has its uses but – as the alternative name suggests – it too has problems.

The second solution is to constrain the parameters passed to a function. This may well be a solution in some contexts, but in the context taken by ENCAPSULATE CONTEXT it is not. In ENCAPSULATE CONTEXT we wish the caller to remain ignorant of what is being passed down. This is a deliberate selective ignorance. While this approach introduces compile time coupling the coupling is less than would be introduced by either global variables or an extra parameter in the function signature.

We could remove this coupling were we to use dynamic members within the context, this is the approach taken by Patow and Lyaret in PARAMETER BLOCK pattern (another pattern presented at EuroPLoP 2003.)

Both Phil and William describe my example as a straw man. To some degree this is true, like any example it is an abstraction which ignores some elements, however, I can assure them it is not a straw man example.

The example given comes from a very real system. Two explanations are therefore possible:

**Option #1:** A better design was possible that would have avoided this design situation. Undoubtedly other designs where possible, would other designs have avoided this problem? I don't know. I do know that this solution troubled me as it emerged; this was the trigger for deeper investigation and ultimately writing the pattern.
More importantly, like any design this evolved, knowing what one knows at the end of a design one might not always chose the same design again. However, I am convinced that given the constraints at the time (knowledge of domain, time to market, programming interfaces, experience, etc.) this was the best design possible.

**Option #2:** Simply that I am a poor designer. I will let readers decide this for themselves but I will note that others have come to similar conclusions in similar circumstances.

Finally, I welcome this debate, it's good to hear other views and it demonstrates that software development is not black and white, different opinions exists and always will. I look forward to hearing more comments and opinions.

*Allan Kelly*
allan@allankelly.net

## Alan's reply to Allan

Allan is right: if a "letter to the editor" had expressed the views that Phil propounded then I would have sought a response from the author. For that I have to apologise, both to Allan and to the readership. Sorry.

Phil's comments caused discussion within the editorial team – not so much because of the views expressed, but how to deal with them. Given Phil's position as one of Overload's advisors his comments needed different treatment to those of most correspondents. Especially as the main point was regarding editorial policy: should material that an advisor has concerns about be published without warning?

My experience with the solution Allan proposes is that a number of problems experienced on the project that I employed it on were resolved without unexpected effects. Hence, I didn't feel it appropriate to give any warning beyond those Allan himself provides.

I trust that Overload readers are sufficiently sophisticated to make up their own minds about both the validity of Allan's pattern paper and also about Phil's concerns regarding publishing it.

*Alan Griffiths (editor)*

## Developing a Pattern

Dear Editor,

I was a little surprised and more than a little worried by the comments made by Phil Bass (included in the editorial of Overload 64) on the ENCAPSULATE CONTEXT pattern by Allan Kelly (published in Overload 63). They were quite strongly against Allan's article, which is a personal preference anyone is allowed to express, but for reasons that I felt were ungrounded, and which undermined the value of those comments as a review.

The first concern I had was with Phil's question over the validity of the pattern. Given Phil's experience and the way that he has approached design in his articles, this is surprising. The solution outlined in the pattern is both valid and good for the problem it addresses, and a standard tool in the toolkit of experienced OO developers. For the record, and to assuage Phil's concerns, it can be found in a number of well-designed systems; it can also be found missing in several systems that are not so well designed, where

instead single, fixed points of contact (globals, SINGLETONS, MONOSTATES, etc) are employed or long, unstable and tedious argument lists are passed around.

Unifying sets of distinct items, such as all or part of an argument list, that are bound by common use or an invariant is one of the diverse range of techniques OO developers use to identify object types. Phil considers such a use to be trivial, which appears to go against much of the accumulated wisdom on identifying stable elements in a design. Such normalisation is a common and accepted practice, whether carried out explicitly or intuitively.

The second concern I had is concerned with a fundamental part of the pattern concept. A pattern is not an unconditional piece of design advice, a blanket recommendation that covers all designs. Its applicability is very much dependent on context and the acceptability of trade-offs involved in applying it. A robustly written pattern will make clear that it is not a panacea, publicising its benefits, liabilities and alternatives quite openly. This is a point that Mark Radford made clearly and well in his editorial of Overload 63. Hence the reason for my surprise: Phil's comments suggest that he views the pattern as unconditional design advice that has neither a discussion of consequences nor a discussion of techniques involved in applying it, such as partitioning the context. I don't believe that this is the way that Phil actually views patterns, but that is the message that comes across in this instance.

Allan's pattern is quite clear in its caution, making explicit the many design decisions and alternative paths that would lead to or away from encapsulating the execution context of an object. Phil does not seem to have picked up on some of the other points made in the paper, such as partitioning the execution context. He proceeds to misapply the pattern in his discussion, and then claim that it is the pattern that is broken and not his reading of it. It is important to recognise that the pattern does not unconditionally propose that there be only one context type or context object (or, to pick up a previous point a conditional application, that context objects are needed in all system designs). To claim that the system's coupling will rise if ENCAPSULATED CONTEXT is applied both misses and makes the point: employ the pattern to reduce the coupling rather than increase it; if that doesn't work, do it differently or do something else. If Phil does not wish to apply the pattern, I have no problem with that; if he has been unable to evaluate it on its own terms and is cautioning others that it should not be used at all, I question that.

The first two concerns are technical in character, and are the stuff of lively technical debate, whatever views we hold. However, the third concern I had was more ad hominem in nature. Phil's claim that "the mention of Kevlin Henney, Frank Buschmann and EuroPLoP gives the article an unwarranted air of authority" is an inappropriate claim that potentially insults all those involved – Allan, Frank, me and the EuroPLoP workshop participants who offered Allan feedback on his paper.

In his prologue and his acknowledgements section Allan offers an insight into the history and evolution of the paper. In recognising that it has evolved, and the journey taken so far, he mentions those who have contributed in some way to the paper. That is part of telling the story of the paper, but it is also why "acknowledgements" sections are so called. Our names were not picked out of thin air to aggrandise Allan's work: we offered Allan feedback in one form or another. It is generally considered polite to acknowledge such contributions. Such acknowledgements are also part of the cultural expectation for pattern papers.

It is perhaps worth understanding a little more about the process that surrounds the reviewing of many pattern papers. A shepherd is someone who offers comments structured as iterative feedback with dialogue, with the goal of giving the author a different insight into their paper and the means to improve it. This is the role that I took on voluntarily when the pattern originally came up in discussion in 2002. At this point Allan was working on the paper without a distinct publication goal in mind. Allan decided to submit it to EuroPLoP 2003, and at this point Frank Buschmann took on the role of the shepherd. Papers are not accepted for the PLoP family of conferences without some amount of shepherding and acceptance by the shepherd. The shepherding is intended to improve the paper to a point where it can be opened to further structured feedback at the conference in the form of a workshop, which is where Allan received further suggestions for improvement. Publication in the EuroPLoP proceedings is not a mark of perfection, but is a visible outcome of the reviewing process, and to reference it is a statement of fact rather than an appeal to authority.

Given the depth and breadth of the review process, it seems only good manners to include a list of acknowledgements. And, given the common characterisation of a pattern paper as a work that is always in progress, it is possible that Allan will take Phil's comments on board and address them in some way. In such a case, it is also likely that Phil's name will appear in the list of acknowledgements, because in one way or another he will have contributed to improving the paper.

*Kevlin Henney*
kevlin@curbralan.com

## Phil's Response

Dear Editor,

The only comment I really wish to make is that I apologise unreservedly to Allan, Kevlin and anyone else who feels insulted by my remarks. When I said that "the mention of Kevlin Henney, Frank Buschmann and EuroPLoP gives the article an unwarranted air of authority" I chose my words badly. I still believe Allan's article over-states the value of this "pattern", but I never intended to question anyone's integrity.

The real technical issue, here, lies with Allan's initial premise that "A system contains data, which must be generally available to divergent parts of the system". That is a description of a problem. ENCAPSULATE CONTEXT is a sticking plaster that can be applied if you wish, but it doesn't begin to tackle the problem itself. What we should be doing is analysing the system's design with a view to removing (as far as possible) the need for such data.

*Phil Bass*
phil@stoneymanor.demon.co.uk

## C++ Lookup Mysteries

Dear Editor,

Sven Rosvall's "C++ Lookup Mysteries" in Overload 63 couldn't have been better timed as it provided a solution to a problem I had been struggling with – a test harness that failed to compile after a new feature was added to the main product because of C++'s non-intuitive name lookup rules.

The problematic code, trimmed to the minimum to illustrate the problem, was:

```
template<class C>
void DebugPrint(const string& description,
                const C& container) {
  cout << description << "\n";
  // some stuff
  copy(container.begin(), container.end(),
       ostream_iterator<typename C::iterator
                  ::value_type>(cout, " "));
  cout << endl;
  // some other stuff
}
```

This works fine for standard containers containing either built-in types or user defined types that define an output operator in the same namespace as the type is defined. The code that broke the test harness was a standard container of `std::pair`. The obvious solution, defining `operator<<(std::pair)` in the test harness namespace, didn't work because the compiler cannot "see" this definition. The problem is that `operator<<` is already defined (for the built-in types) in `namespace std` and masks my definition, as Sven explains "Firstly, the nearest enclosing namespace is searched for 'entities' with the same name. *Note that as soon as a name is found the search stops*" (my italics). C++ name lookup says that the only place that the compiler will look for `operator<<(std::pair)` is in `std`.

Ah, so all I have to do is define it in `std`:

```
namespace std {
  template<class T1, class T2>
  std::ostream& operator<<(std::ostream& os,
                  const std::pair<T1,T2>& p) {
    return os << "(" << p.first
              << "," << p.second << ")";
  }
}
```

except that adding declarations or definitions to `namespace std` is undefined behaviour according to the standard (Clause 17.4.3.1).

Of course the name lookup will find `operator<<(pair)` in the test harness namespace for a `pair` also in that namespace. The standard fully defines `std::pair` (Clause 20.2.2) so I can copy the source code to define my own `pair` (in my workspace) and expect identical behaviour. Although legal, this has a number of problems:

- It breaks the rule of least surprise. A future maintainer may wonder why there are two identical `pairs` in different namespaces.
- It requires the main product source code to use this new `pair`, and thus adds a dependency on the test harness code.
- Although the two `pairs` are binary compatible, they are not interchangeable in source code without considerable scaffolding, and even then not fully.

To force the name lookup to find my `operator<<(std::pair)` without duplicating `std::pair`, I took Sven's wrapper class, `PrintSpannerNameAndGap`, and made it into a template class and output function:

```
template<class T>
class osformatter {
public:
  osformatter(const T& t) : t_(t) {}
  void print(std::ostream& os) const {os << t_;}
private:
  const T & t_;
};

template<class T>
std::ostream& operator<<(std::ostream& os,
                    const osformatter<T>& f) {
  f.print(os);
  return os;
}
```

and changed the line in the debug function to use it:

```
copy(container.begin(), container.end(),
  ostream_iterator<osformatter<typename
     C::iterator::value_type> >(cout, " "));
```

This now works for all built-in types and any user defined types that define an `operator<<` in the same namespace.

Now it is possible to write a specialisation of `osformatter` for any type that does not support the output operator or for which we want some special formatting, for example, fixed precision doubles:

```
class osformatter<double> {
public:
  osformatter(const double& d) : d_(d) {}
  void print(std::ostream& os) const {
    int p=os.precision();
    os.precision(4);
    os << d_;
    os.precision(p);
  }
private:
  const double& d_;
};
```

I can now apply the same specialisation to `std::pair`, which, being a template itself, needs a template declaration for the types contained within it:

```
template<class T1, class T2>
class osformatter<std::pair<T1,T2> > {
public:
  osformatter(const std::pair<T1,T2>& p)
                                  : p_(p) {}
  void print(std::ostream& os) const {
    os << "(" << p_.first
       << "," << p_.second << ")";
  }
private:
  const std::pair<T1,T2>& p_;
};
```

This now compiles because the `std::pair` output code is contained in osformatter, and thus explicitly called, and so is no longer dependent on the name lookup rules.

This not only solved my name lookup problem but provided a nice way of changing the default output format of built-in types when using `copy`.

Regards,

*Mark Easterbrook*
mark@easterbrook.org.uk

# From Waterfall to EVO in a Medium Size Norwegian Software House

## The path and experiences

by Trond Johansen

## Background

FIRM was established in 1996, and has 70 employees in 4 offices (Oslo, London, New York and San Francisco). FIRM delivers one software product: *Confirmit*. Confirmit is a web-based application which enables organizations to gather, analyze and report key business information across a broad range of commercial applications. Confirmit can be applied to any information-gathering scenario. Its three main data sources are: Customer Feedback, Market Feedback and Employee Feedback.

The FIRM R&D department consists of about 20 people, including a Quality Assurance department of 3 people where I work. We are mainly involved in product development of Confirmit, but we also do custom development for clients who fund new modules of the software.

In the very beginning, when FIRM only had a couple of clients, our development was very ad-hoc and customer driven. The software was updated on an almost daily basis based on client feedback. As our client base grew, we formalised the development process according to a waterfall model. We were unhappy with several aspects of the model: risk mitigation postponed until late stages, document-based verification postponed until late stages, attempting to stipulate unstable requirements too early, operational problems discovered too late, lengthy modification cycles and much rework. The requirements were focused on functionality, not on quality attributes.

FIRM CTO Peter Myklebust and I heard Tom Gilb speak about Evolutionary Project Management [EVO] at a software conference ( ITPro 2003). We found the ideas very interesting, and Tom and Kai Gilb offered to give a more detailed introduction to the concept. They spent one day in our offices teaching and preaching EVO. We decided to use EVO as best as we could for the next release, with a development phase of 3 months.

## FIRM's Interpretation of EVO: Basis for the 3 Month Trial Period

EVO is in short: *Quickly evolving towards stakeholder values & product qualities, whilst learning through early feedback.*

After the one day crash course with Tom and Kai Gilb and a literature study ("Competitive Engineering" by Tom Gilb and other material on the subject), our overall understanding of EVO was this:

- Find stakeholders (End users, super-users, support, sales, IT Operations etc)
- Define the stakeholders' real needs and the related Product Qualities
- Identify past/status of product qualities and your goal (how much you want to improve)
- Identify possible solutions for meeting your goals
- Develop a step-by-step plan for delivering improvements with respect to Stakeholder Values & Product Quality goals:
  - Deliveries every week
  - Measure: are we moving towards our goals?

## Requirements

With EVO, our requirements process changed. Previously we focused mostly on function requirements, and not on quality requirements. It is the quality requirements that really separate us from our competitors. There is an analogy with the spell checker in MS Word: why was this a killer application? There was no new functionality; authors of documents have been able to spell check with paper dictionaries for ages. The real difference was a superior product quality: speed of spell checking and usability. *[Surely MS-Word wasn't the first WP with automated spell checking? Ed.]*

We tried to define our requirements according to a basic standard:

- Clear & Unambiguous
- Testable
- Measurable
- No Solutions (Designs)
- Stakeholder Focus

**Example:**

***Usability.Productivity***

**Scale:** Time in minutes to set up a typical specified Market Research Report (MR)

**Past:** 65 min, Tolerable: 35 min, Goal: 25 min (end result was 20 min)

**Meter:** Candidates with knowledge of MR-specific reporting features performed a set of predefined steps to produce a standard MR Report. (The standard MR report was designed by Mark Phillips, an MR specialist at our London office)

The focus is here on the day-to-day operations of our MR users, not a list of features that they might or might not like. We *know* that increased efficiency, which leads to more profit, will please them.

After one week we had defined more or less all the requirements for the next version of Confirmit.

## Solutions/Designs

For every quality requirement we looked for possible solutions (Design Ideas)

E.g. for Quality Requirement: Usability.Productivity we identified the following design ideas:

- DesignIdea.Recoding (See IET below)
- DesignIdea.MRTotals
- DesignIdea.Categorizations
- DesignIdea.TripleS
- ...and many more

We evaluated all these, and specified in more detail those we believed would add the most value (take us closer to the goal).

## EVO

We collected the most promising solutions/design ideas and included them in an Impact Estimation Table (IET) – see Figure 1.

The IET is our tool for controlling the qualities and deliver improvements to real stakeholders, or as close as we can get to them. (E.g. ProS/Support department acting as clients)

## FIRM EVO Week

We decided that one EVO step should last one week (see Table 1) because of practical reasons, even though we violate the rule of not spending more than 2% of project schedule in each step.

| | A | B | C | D | E | F | G | BX | BY | BZ | CA |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | | | | | |
| 2 | | Current Status | Improvements | | Goals | | | Step9 | | | |
| 3 | | | | | | | | Recoding | | | |
| 4 | | | | | | | | Estimated impact | | Actual impact | |
| 5 | | Units | Units | % | Past | Tolerable | Goal | Units | % | Units | % |
| 6 | | | | | Usability.Replacability (feature count) | | | | | | |
| 7 | | 1,00 | 1,0 | 50,0 | 2 | 1 | 0 | | | | |
| 8 | | | | | Usability.Speed.NewFeaturesImpact (%) | | | | | | |
| 9 | | 5,00 | 5,0 | 100,0 | 0 | 15 | 5 | | | | |
| 10 | | 10,00 | 10,0 | 200,0 | 0 | 15 | 5 | | | | |
| 11 | | 0,00 | 0,0 | 0,0 | 0 | 30 | 10 | | | | |
| 12 | | | | | Usability.Intuitiveness (%) | | | | | | |
| 13 | | 0,00 | 0,0 | 0,0 | 0 | 60 | 80 | | | | |
| 14 | | | | | Usability.Productivity (minutes) | | | | | | |
| 15 | | 20,00 | 45,0 | 112,5 | 65 | 35 | 25 | 20,00 | 50,00 | 38,00 | 95,00 |
| 20 | | | | | Development resources | | | | | | |
| 21 | | | 101,0 | 91,8 | 0 | | 110 | 4,00 | 3,64 | 4,00 | 3,64 |

**Figure 1: Impact Estimation Table**

At the Project Management meetings on Fridays each project leader presented the results from the previous step (IET), as well as the content of next EVO step (one week). Possible new Solutions are discussed and weighed against each other.

We launched our first major release based on EVO in May 2004 and we have already received feedback from users on some of the leaps in product qualities. E.g. the time for the system to generate a complex survey has gone from 2 hours (of waiting for the system to do work) to 20 seconds!

## Internal Feedback on EVO After the Trial Period

**Project leaders:**

1. Defining good requirements can be hard.
2. It can be hard to find meters which were practical to use, and at the same time measured real product qualities.
3. Sometimes you think it's necessary to spend more than a day on designs, but this was not right according to our understanding of EVO – the concept of backroom[1] activity was new to us.
4. Sometimes it takes more than a week to deliver something of value to the client – again, the concept of backroom activity was new to us.

**Team members (developers):**

1. Sometimes it felt like we're rushing to the next weekly step before we had finished the current step.
2. Testing was sometimes postponed in order to start next step, and some of these mistakes were not picked up in later testing.

Overall, the whole organization has embraced EVO. We all think it has great potential, and we will work hard to utilize it to the full.

*Trond Johansen*
trond.johansen@firmglobal.com

1 A backroom activity is programming activity not visible for the end user. This is not essential information, we seldom use this activity. We always try to produce some value to some stakeholders every week.

| | Development Team | Users (PMT, Pros, Doc writer, other) | CTO (Sys Arch, Process Mgr) | QA (Configuration Manager & Test Manager) |
|---|---|---|---|---|
| **Friday** | • PM: Send Version N detail plan to CTO + prior to Project Mgmt meeting<br>• PM: Attend Project Mgmt meeting: 12-15<br>• Developers: Focus on general maintenance work, documentation | | • Approve/reject design & Step N<br>• Attend Project Mgmt meeting: 12-15 | • Run final build and create setup for Version N-1<br>• Install setup on test servers (external and internal)<br>• Perform initial crash test and then release Version N-1 |
| **Monday** | • Develop test code & code for Version N | • Use Version N-1 | | • Follow up CI<br>• Review test plans, tests |
| **Tuesday** | • Develop test code & code for Version N<br>• Meet with users to discuss action taken regarding feedback from Version N-1 | • Meet with developers to give feedback and discuss action taken from previous actions | • System Architect: review code and test code | • Follow up CI<br>• Review test plans, tests |
| **Wednesday** | • Develop test code & code for Version N | | | • Review test plans, tests<br>• Follow up CI |
| **Thursday** | • Complete test code & code for Version N<br>• Complete GUI tests for Version N-2 | | | • Review test plans, tests<br>• Follow up CI |

**Table 1: An EVO Step**

# Multiple Streams Going Nowhere

by Paul Grenyer

In this case study I am going to describe two streams I developed for use within my C++ testing framework, Aeryn [Aeryn]. Aeryn has two output streams. One is minimal and only reports test failures and the test, pass and failure counts. The other is more verbose and includes all the output from the minimal stream, plus a list of all test sets along with their individual test cases. The minimal stream is intended to be sent to the console and the verbose stream to a more permanent medium such as a log file or database, but either can be sent to any sort of output stream.

The use of the two streams introduces two specific problems:

1. The stream sink for both streams must be passed into the function that runs the tests. For example:
   ```
   std::ofstream verbose("testlog.txt");
   std::stringstream minimal;
   testRunner.Run(verbose, minimal);
   ```
   Even if only one of the two outputs is required, both streams must be specified.

2. The same information must be sent to both streams, which results in duplicate code. For example:
   ```
   verbose << "Ran 6 tests, 3 passes, 3 failures";
   minimal << "Ran 6 tests, 3 passes, 3 failures";
   ```
   This is far from ideal as every time the text sent to one stream is modified, the text sent to the other stream must also be modified. It would be all too easy to forget to update one or other of the streams or to update one incorrectly.

Both of these problems can be solved by writing a custom stream. Writing custom streams is covered in detail in section 13.13.3 (User-Defined Stream Buffers) of The C++ Standard Library [Josuttis]. As Josuttis does such a good job of describing custom streams and his book is widely distributed, I will only cover the necessary points relevant to this case study.

## Null Output Stream

Problem 1 can be easily solved with a null output stream. A null output stream is a type of null object [Null Object]. Kevlin Henney describes a null object as follows: "The intent of a null object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behaviour." So basically a null output stream is a stream that does nothing with what is streamed to it. Therefore if either the minimal or verbose stream is not required it can be directed to a null output stream. For example:

```
cnullostream ns;
testRunner.Run(ns, std::cout);
```

The key to writing a custom stream is implementing its stream buffer. The functionality for stream buffers is held in the standard library template class `std::basic_streambuf`. Custom stream buffers can be written by inheriting from `std::basic_streambuf` and overriding the necessary member functions.

It is not necessary for the custom stream buffer to be a template, but it makes life a lot easier if you want your custom stream to work with `char`, `wchar_t` and custom character traits. This is also discussed in detail in The C++ Standard Library.

```
template<typename char_type, typename traits
               = std::char_traits<char_type> >
class nulloutbuf : public
     std::basic_streambuf<char_type, traits> {
protected:
  virtual int_type overflow(int_type c) {
    return traits::not_eof(c);
  }
};
```

The code above shows the complete implementation for the null output stream buffer. The `overflow` member function is all that is needed to handle characters sent to the stream buffer. The `traits::not_eof(c)` function ensures that the correct character is returned if `c` is `EOF`.

Now that the stream buffer is complete it needs to be passed to an output stream. The easiest way to do this is to inherit from `std::basic_ostream` and have the stream buffer as a member of the subclass.

```
template<typename char_type, typename traits
               = std::char_traits<char_type> >
class null_ostream : public
       std::basic_ostream<char_type, traits> {
private:
  nulloutbuf<char_type, traits> buf_;

public:
  null_ostream()
    : std::basic_ostream<char_type,
                    traits>(&buf_), buf_() {}
};
```

Notice the constructor initialisation list. The `buf_` member of `null_ostream` is passed to the `basic_ostream` base class before it has been initialised. In his book Josuttis actually puts `buf_` first in the list, but this makes no difference. The base class is still initialised before `buf_`. This could give rise to a problem where `buf_` is accessed by a `nullstream` base class prior to it being initialised.

Some standard library implementations do nothing to avoid this and they don't need to. A library vendor knows their own implementation and if protection was required it would be provided. As the C++ standard gives no guarantee it is sensible for a custom stream to take steps to avoid the stream buffer being accessed before it is created. One way to do this is to put it in a private base class, which is then initialised before `basic_ostream`:

```
template<typename char_type, typename traits>
class nulloutbuf_init {
private:
  nulloutbuf<char_type, traits> buf_;

public:
  nulloutbuf<char_type, traits>* buf() {
    return &buf_;
  }
};
```

```
template<typename char_type, typename traits
                = std::char_traits<char_type> >
class nullostream : private virtual
        nulloutbuf_init<char_type, traits>,
      public
        std::basic_ostream<char_type, traits> {
private:
  typedef nulloutbuf_init<char_type, traits>
                              nulloutbuf_init;

public:
  nullostream() : nulloutbuf_init(),
          std::basic_ostream<char_type,
            traits>(nulloutbuf_init::buf()) {}
};
```

The code above shows that as well as being inherited privately, `nulloutbuf_init` is also inherited virtually. This makes sure that `nulloutbuf` and `nulloutbuf_init` are initialised first, avoiding the undefined behaviour described in 27.4.4/2 of the [C++ Standard]. The undefined behaviour would occur if `nulloutbuf`'s constructor was to throw in between the construction of `basic_ios` (a base class of `basic_ostream`) and the call to `basic_ios::init()` from `basic_ostream`'s contrustor. See the C++ standard for more details.

Now that the implementation of `null_ostream` is complete two helpful typedefs can be added. One for `char` and one for `wchar_t`:

```
typedef null_ostream<char> cnullostream;
typedef null_ostream<wchar_t> wnullostream;
```

I always like to unit test the code I write and usually the tests are in place beforehand. Naturally I use Aeryn for unit testing. Testing `null_ostream` has its own interesting problems. I started by writing two simple tests to make sure that `cnullostream` and `wnullostream` compile and accept input:

```
void CharNullOStreamTest() {
  cnullostream ns;
  ns << "Hello, World!" << '!' << std::endl;
}

void WideNullOStreamTest() {
  wnullostream wns;
  wns << L"Hello, World!" << '!' << std::endl;
}
```

The whole point of a null output stream is that it shouldn't allocate memory when something is streamed to it; otherwise something like a `std::stringstream` could be used instead. Wanting to test for memory allocation caused me to write, with considerable help from `accu-general` members, a memory observer library, called Elephant (see sidebar) [Elephant]. Elephant allows me to write an observer (`NewDetector`) which can detect allocations from within `null_ostream`'s header file, which in this case, also holds its definition. Originally the observer was intended to monitor all allocations that occurred while using `null_ostream`, but as the standard permits stream base classes to allocate memory to store the current locale, I restricted the observer to allocations from `null_ostream` itself:

```
class NewDetector : public
elephant::IMemoryObserver {
private:
  bool memoryAllocated_;

public:
  NewDetector()
    : memoryAllocated_(false) {
  }

  virtual void OnAllocate(void*, std::size_t,
                          std::size_t,
                          const char* file) {
    // Crude black list.
    if(std::strcmp(file,
                   pg::null_ostream_header)) {
      memoryAllocated_ = true;
    }
  }

  virtual void OnFree(void*) {}

  bool AllocationsOccurred() const {
    return memoryAllocated_;
  }
};
```

In order to get `OnAllocate` to be called by the Elephant `operator new` overload that includes the name of the file it was called from, a macro must be introduced into `null_ostream`'s definition. The easiest way to do this is to wrap `null_ostream`'s header file with the macro in the test source file:

```
// nullostreamtest.h
#define new ELEPHANTNEW
#include "nullostream.h"
#undef new
```

## Elephant: C++ Memory Observer

A full discussion of the design of Elephant is beyond the scope of this case study, but the principles on which it is based are simple and easy to explain. Elephant consists of two main components:

**`new` / `delete` overloads:** Elephant has a total of eight pairs of `new` / `delete` overloads. As well as allocating and freeing memory, each overload registers its invocation with the memory monitor by passing the address of the memory that has been allocated or freed. Four of the eight `new` overloads also pass the line and file from which `new` was invoked.

**Memory monitor:** Calls to the `new` / `delete` overloads are monitored by the memory monitor. The memory monitor is observer-compatible and users of Elephant can write custom observers (or use those provided) and register them with the memory monitor. Every time memory is allocated or freed via the `new` / `delete` overloads each observer is notified and passed the memory address and, where available, the line and file from which `new` was invoked.

In order to make sure that `OnAllocate` only registers allocations from `null_ostream`, a variable must be introduced into `null_ostream`'s header file:

```
const char* const null_ostream_header
                                    = __FILE__;
```

An ideal solution would not require the `null_ostream` header to be modified at all for testing. However I could not find a satisfactory alternative. Suggestions will be gratefully received.

Moving `CharNullOStreamTest` and `WideNullOStreamTest` into a class, and giving them new names to better represent what they now test for, allows `NewDetector` to be added as a member, and using Aeryn's `incarnate` function allows a new instance to be created for each test function call.

```
class NullOStreamTest {
private:
  NewDetector newDetector_;

public:
  NullOStreamTest()
    : newDetector_() {
    using namespace elephant;
    MemoryMonitorHolder().Instance().
                AddObserver(&newDetector_);
  }

  ~NullOStreamTest() {
    using namespace elephant;
    MemoryMonitorHolder().Instance().
                RemoveObserver(&newDetector_);
  }

  void NoMemoryAllocatedTest() {
    cnullostream ns;
    ns << testString << testChar
       << std::endl;
    IS_FALSE(
          newDetector_.AllocationsOccurred());
  }

  void NoMemoryAllocatedWideTest() {
    wnullostream wns;
    wns << wtestString << wtestChar
        << std::endl;
    IS_FALSE(
          newDetector_.AllocationsOccurred());
  }
};
```

## Multi Output Stream

Problem 2 can be solved with what I have called a multi output stream. A multi output stream forwards anything that is streamed to it onto any number of other output streams. To solve the problem faced by Aeryn the multi output stream could simply hold references to two streams (one verbose, one minimal) as members, but this could potentially restrict future use when more than two streams may be required.

Again, the key is the output buffer. The first element to consider is how the multiple output streams, or at least some sort of reference to them, will be stored and how they will be added to and removed from the store. The easiest way to store the output streams is in a vector of `basic_ostream` pointers.

The original design for the multi output stream I came up with managed the lifetime of the output streams as well. This involved the output streams being created on the heap and managed by a vector of smart pointers. Therefore a smart pointer either had to be written or a dependency on a library such as boost [boost] introduced. As the lifetime of the multi output stream would be the same or very similar to the lifetime of the output streams there was really no need.

The easiest way to add and remove output streams is by way of an `add` function and a `remove` function. This functionality is shown in the code below.

```
template<typename char_type, typename traits
               = std::char_traits<char_type> >
class multioutbuf : public
               std::basic_streambuf<char_type,
                                    traits> {
private:
  typedef std::vector<std::basic_ostream<
                        char_type, traits>* >
                        stream_container;
  typedef typename stream_container::iterator
                        iterator;
  stream_container streams_;

public:
  void add(std::basic_ostream<char_type,
                              traits>& str) {
    streams_.push_back(&str);
  }

  void remove(std::basic_ostream<char_type,
                              traits>& str) {
    iterator pos = std::find(streams_.begin(),
                       streams_.end(), &str);

    if(pos != streams_.end()) {
      streams_.erase(pos);
    }
  }
};
```

The `add` function simply adds a pointer to the specified output stream to the store. The `remove` function must first check that a pointer to the specified output stream exists in the store, before removing it.

Josuttis describes the `std::basic_streambuf` virtual functions that should be overridden in a custom output buffer: `overflow` for writing single characters and `xsputn` for efficient writing of multiple characters.

```
template<typename char_type, typename traits
               = std::char_traits<char_type> >
class multioutbuf : public
               std::basic_streambuf<char_type,
                                    traits> {
  ...
```

```
protected:
  virtual std::streamsize xsputn(
                  const char_type* sequence,
                  std::streamsize num) {
    iterator current = streams_.begin();
    iterator end = streams_.end();

    for(; current != end; ++current) {
      (*current)->write(sequence, num);
    }

    return num;
  }

  virtual int_type overflow(int_type c) {
    iterator current = streams_.begin();
    iterator end = streams_.end();

    for(; current != end; ++current) {
      (*current)->put(c);
    }

    return c;
  }
};
```

A different approach would be to write three function objects and use `for_each` to call the appropriate function for each output stream in the store. However, this would not add a lot to the clarity and would not provide any better performance, but would create a lot of extra code.

The output buffer must be initialised and passed to an output stream and the output stream needs to have corresponding `add` and `remove` functions that forward to the output buffer's functions:

```
template<typename char_type, typename traits>
class multioutbuf_init {
private:
  multioutbuf<char_type, traits> buf_;

public:
  multioutbuf<char_type, traits>* buf() {
    return &buf_;
  }
};

template<typename char_type, typename traits
              = std::char_traits<char_type> >
class multiostream : private
       multioutbuf_init<char_type, traits>,
      public
       std::basic_ostream<char_type, traits> {
private:
  typedef multioutbuf_init<char_type, traits>
                             multioutbuf_init;

public:
  multiostream() : multioutbuf_init(),
         std::basic_ostream<char_type,
           traits>(multioutbuf_init::buf()) {}

  bool add(std::basic_ostream<char_type,
                              traits>& str) {
    return multioutbuf_init::buf()
                              ->add(str);
  }

  bool remove(std::basic_ostream<char_type,
                              traits>& str) {
    return multioutbuf_init::buf()
                              ->remove(str);
  }
};
```

All that remains is to provide two convenient typedefs, one for `char` and one for `wchar_t`:

```
typedef multi_ostream<char> cmultiostream;
typedef multi_ostream<wchar_t> wmultiostream;
```

The multi output stream is quite easy to test and should be tested for the following things:

1. Output streams can be added to the multi output stream.
2. All added output streams receive what is sent to the multi output stream.
3. Streams can be removed from the multi output stream.

Although this looks likes three separate tests they are all linked and the easiest thing to do is to write a single test for `char`:

```
void CharMultiOStreamTest() {
  std::stringstream os1;
  std::stringstream os2;

  cmultiostream ms;
  ms.add(os1);
  ms.add(os2);
  ms << "Hello, World";

  IS_EQUAL(os1.str(), "Hello, World");
  IS_EQUAL(os2.str(), "Hello, World");

  ms.remove(os1);
  ms << '!'

  IS_EQUAL(os1.str(), "Hello, World");
  IS_EQUAL(os2.str(), "Hello, World!");
}
```

and a single test for `wchar_t`:

```
void WideMultiOStreamTest() {
  std::wstringstream wos1;
  std::wstringstream wos2;

  wmultiostream wms;
  wms.add(wos1);
  wms.add(wos2);
  wms << L"Hello, World";
```

[concluded at foot of next page]

# Stable Intermediate Forms
## A Foundation Pattern for Derisking the Process of Change
**by Kevlin Henney**

*The universe is change; life is what thinking makes of it.*
Marcus Aurelius

Change is often associated with risk, in particular the risk of failure or unwanted side effects. Whether such change is the move from one position on a rock face to another during a climb, or the progress of development through a software project, or the change of state within an object when it is being copied to, risk is ever present. In all cases, change is associated with questions of confidence, consequence and certainty. In the event of any failure or unknowns, change can cause further failure and greater unknowns – a missed foothold, a missed deadline, a missed exception.

Stable Intermediate Forms is a general pattern for making progress with confidence and limiting the effect of failure. When undertaking a change or series of changes to move from one state to another, rather than carrying out a change suddenly and boldly, with a single large stride but uncertain footfall, each intermediate step in the process of change is itself a coherent state. This high-level pattern manifests itself in various domain-specific patterns, three of which are reported informally in this paper and used as examples: Three Points of Contact for rock climbing; Iterative and Incremental Development for software development process; and Copy Before Release for exception safety in C++. It is a matter of preference and perspective as to whether this paper is considered to document one, three or four patterns.

### Thumbnail

Change typically involves risk. Any change from one state of affairs to another that cannot be characterised as atomic inevitably involves a number of steps, any one of which could fail for one reason or another, leaving the change incomplete and the circumstances uncertain. Therefore, ensure that each intermediate step in the process of change expresses a coherent state, one that in some meaningful way represents a whole rather than a partial state of affairs.

### Example: Three Points of Contact

There you are, halfway up a cliff. Your face pressed against the rock face. Gravity is carefree but ever present. The ground is unforgiving and ever distant. The rock is your friend. And you hope that your friend, standing on the ground below you, is like a rock. In the event of a slip, his belaying should provide you with the failsafe that prevents you from joining him too rapidly.

To go up, you need to go right and then up. To do this you need to reach for what promises to be a good hold, but it is a little too far away to reach comfortably. There is no convenient ledge or outcrop to rest your whole weight on, and the rock is both sheer and wet. You recall that really cool bit at the beginning of *Mission Impossible 2* where free-climbing Tom Cruise, presented with a challenge, leaps from one rock face to another and manages, with a little grappling, to secure himself. Very cool... but not really an option: (1) Tom Cruise loses his grip and only just manages to avoid falling; (2) you are not in a movie; (3) you are the lead climber, so there is still a little way to fall and some significant rock to bounce off before the rope becomes taut and carries your weight; and (4) you are a relatively inexperienced climber and new to leading, so experience is less likely to be on your side.

Alternatively, you can try leaning over, with one foot jammed in the crack you are currently using as a handhold, and reaching for the new hold with your right hand as you angle to the right. There is nothing substantial that your left hand can hold once you have stretched to the right and the other foot would have to be free, providing balance but not support. Should work, if you have the distance right (being a little taller at this point would have helped...), but not exactly a sure thing.

However, there is yet another way. By dropping down a little, reversing some of your ascent, and then climbing across and then up, you could do it without either the dramatics or the uncertain

---

[continued from previous page]

```
    IS_EQUAL(wos1.str(), L"Hello, World");
    IS_EQUAL(L"Hello, World", wos2.str());

    wms.remove(wos1);
    wms << '!';

    IS_EQUAL(L"Hello, World", wos1.str());
    IS_EQUAL(L"Hello, World!", wos2.str());
}
```

### Conclusion

Streams are a hugely powerful part of the C++ language; which few people seem to make use of and even fewer people customise for their own uses. The `null_ostream` and `multi_ostream` are very simple examples of customisation and I have shown here just how easy stream customisation is.

*Paul Grenyer*
paul@paulgrenyer.co.uk

### References

[Aeryn] *Aeryn: C++ Testing Framework.* http://www.paulgrenyer.co.uk/aeryn/

[Josuttis] Nicolai M. Josuttis, *The C++ Standard Library*, Addison-Wesley, ISBN: 0-201-37926-0.

[Null Object] Kevlin Henney. *Null Object, Something for Nothing*, http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/NullObject.pdf

[C++ Standard] *The C++ Standard*, John Wiley and Sons Ltd. ISBN: 0-470-84674-7.

[Elephant] *Elephant: C++ Memory Observer*: http://www.paulgrenyer.dyndns.org/elephant/

[Boost] *Boost*. http://www.boost.org/

### Acknowledgements

Thank you to Jez Higgins, Alan Stokes, Phil Bass, Alan Griffiths, Alisdair Meredith and Kevlin Henney for their comments at various stages of this case study.

leaning. There are enough handholds to afford you three points of contact most of the way, ensuring that you can shift your weight more easily across the rock face – two feet and one hand or two hands and one foot providing support at all times. Although it'll take another minute there is a much greater chance of success and much greater certainty that it will be only a minute.

## Example: Iterative and Incremental Development

There are a number of different activities involved in taking a software development project from inception through to deployment, and a project will pass through a number of distinct phases. Whether it is the comprehension of what needs to be built, the formulation of an architectural vision, the writing of code, the running of tests, and so on, there is a question of how the activities should be played out over time, how they relate to one another, and how they relate to the planned, sequential phases of a project.

The manufacturing metaphor of software development has a certain simple attraction. Activities are strictly aligned with phases, so that comprehension of the problem domain is done during a phase dedicated to understanding the problem domain, coding is done during a phase of the project dedicated to writing code, and so on. Because each phase is carried out in sequence, each activity is also carried out in sequence, making the development a pipeline.

However, it is an idealised pipeline, where the output of one activity becomes the input to the next, without a feedback loop, and the pipeline is assumed to be non-lossy and free from interference. These assumptions are somewhat difficult to replicate in the real world. The pipeline is shielded – or rather, blinkered – from the business of dealing with clients, responding to change and clarification, or working with and learning from colleagues. Risks, and the consequent element of surprise they bring to a schedule, tend to stack up rather than drop down as development proceeds. There is no adequate mechanism for handling additional requirements, clarification of requirements, changes in staff or organizational structure, shifts in objectives, as well as technical issues and unknowns. The inevitability of any one of these automatically jeopardises the assumptions that might make a pipeline a valid and viable solution.

Pipeline approaches are exemplified by the conventional view of the Waterfall Development Lifecycle, a term that conjures up a majestic image, albeit one that ends with water crashing against rocks with great force. The predicted and steady progression from *analysis*, through *design*, into *implementation*, *testing*, and then finally *deployment*, has an undeniable charm and simplicity. A pipeline project plan looks great on paper. It is easy to understand. It is easy to explain. It is easy to track. It is easy to fall under its spell. The fact that it may bear little relation to how people work or the actual progress of a project may seem somehow less important when caught in its thrall.

There is an implicit assumption that when a project deviates from its pipelined schedule it is due to some concrete aspect of the development – the developers, the management, the optimism of the schedule, external factors – rather than a fault in the underlying development metaphor. The reaction is often to declare that "we'll do it right next time" and then throw all hands – and more – to the pump to push through the delivery – more staff, more money, more time, less testing, less attention to detail, less application of best practice. The innate human response is that lost time can be made up for, that continual requirements clarification and change is an external factor not intrinsic to the nature of software development, and that making the development process more formal "next time" will address the shortcomings of the present.

For it to be of practical use, a development macroprocess should give more than the illusion of order. However, the notion of making something ordered by strictly pigeonholing one activity to one phase is not a well-measured response. A development process should also actively seek to reduce the risks inherent in development rather than ignore or amplify them. Risk tends to accumulate quietly in the shadow of planned development pipelines, bursting at just the wrong moment – inevitably later rather than sooner.

Activities do not need to be aligned discretely and exclusively with phases. Instead, the lifecycle can be realigned so that the activities run throughout. This is not to say that the emphasis on each activity is identical and homogeneous throughout the development, just that each is not strictly compartmentalised. There is likely to be a stronger emphasis on understanding the problem domain early on in the development than later, and likewise a stronger emphasis on deployment and finalization towards the end than at the start, but these different concerns are not exclusive to the beginning and end.

Instead of aligning the development cycle with respect to a single deadline and deliverable at its end, structure it in terms of a series of smaller goals, each of which offers an opportunity for assessment and replanning. Each subgoal should be a clearly defined development increment. However, because usable functionality is not proportional to quantity of code, the completion of code artefacts (lines of code, classes, packages, layers) cannot meaningfully be used as a measure of progress. As indicators they are too introspective and are at best only weakly correlated with accessible functionality. Functionality, whether defined by usage scenarios or feature models, is a more visible indication of software completeness. A development increment defined in terms of functionality is a more identifiable concept for all stakeholders concerned, whether technical or non-technical, whereas code artefacts have meaning only for developers.

Breaking down development objectives into these smaller increments allows all stakeholders to make a more meaningful assessment of progress and provides more opportunities to refine or rearrange objectives for future increments. However, should increment deadlines be fixed or variable? In other words, should development on an increment stop when its initial deadline passes, regardless of the scope that has yet to be covered, or should the deadline be pushed back until the intended scope of the increment has been completed?

Spacing the increment deadlines regularly, whether one week or one month apart, establishes a rhythm that offers a useful indication of progress, answering the question of how much functionality can be covered in a fixed amount of time. The scope is therefore variable and functionality must be prioritised to indicate which features can be dropped if the time does not allow for all intended functionality to be completed. Fixing the scope rather than the time means that the deadline, and therefore the point of assessment or demonstration, is always unknown. It is difficult to schedule people, meetings and subsequent development if dates are always the unknown. Any unexpected problems in the development of an increment can mean that the increment begins to drag on without apparent end, when it would perhaps be better cut short and

subsequent iterations reconsidered. Therefore, fixed scope per increment decreases rather than increases the knowability of development progress, and is therefore a riskier option than establishing a steady pulse and measuring scope coverage to date.

The development of each increment has its own internal lifecycle that is made up of the activities that run through the whole development. The repetition of this mini-lifecycle leads to an iterative process that is repeated across the whole macro-lifecycle. Each iteration has an associated set-up and tear-down cost, but with regularity the effort involved in kick-off and increment finalization is reduced through familiarity.

Iterative and incremental development derisks the overall development process by distributing the risk over the whole lifecycle, rather than towards the end, and making the progress more visible. The difficult-to-obtain determinism of knowing all of the problem before all of the solution, of writing all of the code before all of the tests, and so on, is traded for scheduling determinism and a more certain and empirical exploration of the scope.

## Example: Copy Before Release

Coding in the presence of exceptions is a very different context to coding in their absence. Many comfortable assumptions about sequential code have the rug pulled from beneath them. Writing exception-safe code requires more thought than writing exception-unsafe code. It is possible, if care is not taken, that an object may be left in an indeterminate state and therefore, by implication, the whole program put into an inappropriate state.

When an exception is thrown it is important to leave the object from which it was thrown in a consistent and stable state:

- Ideally any intermediate changes should be rolled back.
- Alternatively the object may be left in a partial state that is still meaningful.
- At the very least the object should be marked and detected as being in a bad way.

The consequence of not taking responsive and responsible action can lead to instability. This is both unfortunate and ironic: exception handling is supposed to achieve quite the opposite!

Exception safety can be attained via one of three paths:

- **Exception-aware code:** Code may be scaffolded explicitly with exception handling constructs to ensure that a restabilising action is taken in the event of an exception, e.g. a finally block in Java or C#.
- **Exception-neutral code:** Code can be written to work in the presence of exceptions, but does not require any explicit exception-handling apparatus to do so, e.g. Execute-Around Object in C++ [Henney2000a] or Execute-Around Method in Smalltalk [Beck1997] or Java [Henney2001].
- **Exception-free code:** Guaranteeing the absence of exceptions automatically buys code exception safety.

Exception-aware code is tempting because it is explicit, and therefore has the appeal of safety by diligence. It makes it look as if measures are being taken to deal with exceptions. However, such explicit policing of the flow is not always the best mechanism for keeping the peace. Exception-free code sounds ideal, but this path makes sense only when there is either no change of state to be made or the change is in some way trivial, e.g. assignment to an int or setting a pointer to null. A forced and uncritical attempt to cleanse code of exceptions leads to control flow that is far more complex than it would be with exceptions – a lot of C code bears witness to the twisted logic and opaque flow involved in playing return-code football. In some senses, forcing exception freedom on code leads to the same growth in supporting logic as the "be vigilant, behave" exception-aware doctrine. As with Goldilocks, when she decided to crash the Three Bears' residence and then crash on their beds, we find that exception-aware code can be too hard, exception-free code can be too soft, but exception-neutral code can be just right.

Of the three paths, it is exception-neutral code that is most often the path of enlightenment. Consider the following C++ example. The code fragment sketches a Handle-Body [Coplien1992, Gamma+1995] arrangement for a value-object type, a class that among other features supports assignment:

```
class handle {
public:
  ~handle();
  handle &operator=(const handle &);
  ....
private:
  class representation {
    ....
  };
  representation *body;
};
```

The memory for the body must be deallocated at the end of the handle's lifetime in its destructor, either by replacing the plain pointer with a suitable smart-pointer type, such as `std::auto_ptr`, or explicitly, as follows:

```
handle::~handle() {
  delete body;
}
```

The default semantics for the assignment operator are shallow copying, whereas a handle-to-handle assignment needs to deep copy the body. The traditional model for writing an assignment operator is found in the Orthodox Canonical Class Form [Coplien1992]:

```
handle &handle::operator=(const handle &rhs) {
  if(this != &rhs) {
    delete body;
    body = new representation(*rhs.body);
  }
  return *this;
}
```

An initial inspection suggests that this implementation has addressed the core needs of correct assignment: there is a check to prevent corruption in the event of self-assignment; the previous state is deallocated; new state is allocated; `*this` is returned as in any normal assignment operator. The construction of the nested body is assumed to be self-contained, but what if the representation constructor were to throw an exception? Or if `new` were to fail and throw `std::bad_alloc`? Here be dragons!

A failed creation will result in an exception: control will leave the function, leaving `body` pointing to a deleted object. This stale pointer cannot then be dereferenced safely: the object is unstable and unsafe. When the handle object itself is destroyed – for instance, automatically at the end of a block – the attempt to `delete` body will likely wreak havoc in the application and let slip the dogs of core.

It is tempting to construct a `try catch` block to guard against exception instability. This exception-aware approach normally leads to quite elaborate and intricate code; the solution often looks worse than the problem:

```
handle &handle::operator=(const handle &rhs) {
  if(this != &rhs) {
    representation *old_body = body;
    try {
      body = new representation(*rhs.body);
      delete old_body;
    }
    catch(...) {
      body = old_body;
      throw;
    }
  }
  return *this;
}
```

It is worth pausing a moment to consider the rococo splendour of this code [Hoare1980]:

> *There are two ways of constructing a software design: One way is to make it so simple that there are* obviously *no deficiencies and the other is to make it so complicated that there are no* obvious *deficiencies.*

Nonetheless, in spite of the ornate opaqueness, exception awareness for safety is still a strongly tempting path to follow. Such a solution reflects a way of thinking about the problem that is too close to the problem: a haphazard solution grown by grafting onto the original exception-unsafe code. A far simpler and exception-neutral solution is found in careful ordering of actions, so that the object's state passes through valid intermediate forms, stable no matter what rude interruption befalls the object:

```
handle &handle::operator=(const handle &rhs) {
  representation *old_body = body;
  body = new representation(*rhs.body);
  delete old_body;
  return *this;
}
```

The essential flow expressed in the COPY BEFORE RELEASE C++ idiom [Henney1997, Henney1998] can be characterised simply:

1 Remember the old state.
2 Copy the new state.
3 Release the old state.

The same sequence of actions can be expressed using another idiomatic C++ form that relies on the handle's copy constructor:

```
handle::handle(const handle &other)
  : body(new representation(*other.body)) {}
```

And on a non-throwing swap function, which allows two objects to exchange their state without the possibility of generating an exception [Sutter2000]. The copy of the object takes the role of an EXECUTE-AROUND OBJECT:

```
handle &handle::operator=(const handle &rhs) {
  handle copy(rhs);
  std::swap(body, copy.body);
  return *this;
}
```

The assignment operator takes a copy of the right-hand side. This may throw an exception, but if it does the current object is unaffected. The representation of the current object and the copy are exchanged. The assigned object now has its new state and the copy now has custody of its previous state, which is cleaned up on destruction at the end of the function.

The general form of COPY BEFORE RELEASE can be found repeated across other idioms for exception safety: checkpoint the current state; perform state-changing actions that could raise exceptions; commit the changes; perform any necessary clean up.

## Problem

Change typically involves risk. Any change from one state of affairs to another that cannot be characterised as atomic inevitably involves a number of steps, any one of which could fail. The effect of successful completion is understood, as is the effect of failure at the start, but what of partial failure? Partial failure can lead to instability and further descent into total failure.

The most direct path from one state to another is most often the one with the strongest appeal and the greatest appearance of efficiency. What needs to be done is clear, allowing a single-minded focus on the path of change free of other distractions. All other things being equal, it will be the shortest and most direct path. However, when all other things are not equal the simple view supported by a direct change becomes a simplistic view lacking in care and foresight. The goal of minimizing overheads trades priorities with the safety net of contingency planning and derisking.

Focusing on one goal to the exclusion of others can weaken rather than strengthen control over risk, undermining rather than underpinning certainty. The polymath Herbert Simon coined the term *satisfice* to describe a form of behavioural satisfaction and sufficiency that accommodates multiple goals [Wikipedia]:

> *In economics, satisficing is behaviour which attempts to achieve at least some minimum level of a particular variable, but which does not strive to achieve its maximum possible value. The most common application of the concept in economics is in the behavioural theory of the firm, which, unlike traditional accounts, postulates that producers treat profit not as a goal to be maximized, but as a constraint. Under these theories, although at least a critical level of profit must be achieved by firms, thereafter priority is attached to the attainment of other goals.*

However, it is not necessarily the case that there is more control or predictability with respect to change simply because progress does not become the sole concern of a change of state. The wrong variables can be taken into account or an overcautious stance can lead to wasted effort.

In an effort to combat the uncertainty of the future, overdesign is a common response [Gibson2000]:

> *That which is overdesigned, too highly specific, anticipates outcome; the anticipation of outcome guarantees, if not failure, the absence of grace.*

Thus, a total and explicit approach to design or planning can become a measure that is overvalued and overused to the exclusion of other quantities and qualities that are potentially more valuable and balancing.

A big-bang approach to change proposes a discontinuous change from one state to another, essentially an abrupt shift in observable phenomena. Although it is easy to identify the point of change, there is significant risk if the change does not go according to plan. By contrast an approach that is chaotic rather than catastrophic emphasises concurrency of change rather than a single point of change. However, as with free-threaded approaches to concurrency in software, change becomes difficult to trace, track and reason about. A more bounded and constrained approach to concurrency compartmentalises this uncertainty.

## Solution

Ensure that each intermediate step in the process of change represents a coherent state of play. The risk and consequences of

catastrophic failure are mitigated by ensuring that each mini-change making up the intended change is itself a well-defined and stable step that represents a smaller risk. The result of any failure will be a fall back to a coherent state rather than a chaotic unknown from which little or nothing can be recovered. Each step is perceived as atomic and each intermediate state as stable.

This sequencing of STABLE INTERMEDIATE FORMS underpins rock climbing, exception safety and many successful software development strategies and tactics, as well as other disciplines of thought and movement, for example, T'ai Chi. The term itself is taken from the following observation by Herbert Simon (as quoted by Grady Booch [Booch1994]):

> *Complex systems will evolve from simple systems much more rapidly if there are stable intermediate forms than if there are not.*

The implication of stable intermediate forms is that change is realised as a gradual, observable and risk-averse process – secure the current situation; prepare for a small change; commit to it; repeat as necessary. Not only does it break change down into more than a single, sudden step, but the steps in between have a stability and natural balance of their own. They are visible and discrete, which means that they can be used and measured without necessarily implementing further change. But although discrete, there is enough continuity to ensure that change can not only be paused, it can also take a new direction, even a reversal, without requiring disproportionate effort to replan and put into effect.

Failure does not unduly upset the system or require disproportionate recovery effort to restore or move it to a sensible state should the step to the next stable form falter. A missed handhold need not result in a fall. A thrown exception need not corrupt the state of the throwing object, nor does it require disproportionate effort to rollback – rollback happens by default.

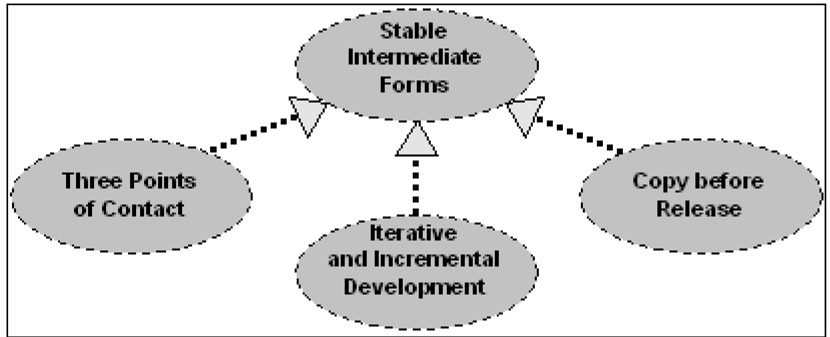
However, it is not necessarily the case that stability should simply be seen as a synonym for good without understanding its potential liabilities and context of applicability. There is a cautiousness, thoroughness and deliberation about STABLE INTERMEDIATE FORMS that is not necessarily appropriate for all systems and objectives. Over a short enough timeframe the progression of small steps may be slower than taking a single leap and, more importantly, slower than is needed to achieve the end goal. For example, with at least one foot on the ground at any one time, walking is an inherently stable and leisurely activity. Running, on the other hand, is inherently unstable, trading raw speed for stability and effort. If speed is the goal a hundred meters is better covered with a run than a walk.

Ascending a familiar route on a climbing wall is derisked with respect to both experience – it is familiar, and therefore not an unknown – and environment – an indoor climbing centre has limitations and facilities that make it a more controlled and comfortable experience than being stuck on a real rock face in the middle of nowhere in bad weather. Other sources of risk mitigation, such as a no-throw exception guarantee or familiarity with a particular climb, can moderate the need for further derisking through STABLE INTERMEDIATE FORMS.

On the other hand, where a prime objective is to embrace rather than mitigate risk – the thrill of a hard climb, the adrenalin rush of a sprint, the intellectual high of tackling a hard problem by immersion in detail, caffeine and the small hours – STABLE INTERMEDIATE FORMS "spoils the fun" and may be seen as inappropriate. Of course, if this prime objective is not shared by others, a failure to progress through stable intermediate forms may be perceived as subjective fancy rather than objective sensibility.

## Discussion

As a pattern, STABLE INTERMEDIATE FORMS may be considered general and abstract, a high-level – even meta-level – embodiment of a principle that recurs in other more concrete, domain-specific patterns where the context of applicability is explicitly and intentionally narrowed to address more specific forces, solution structures and consequences. The three examples in this paper are realizations of STABLE INTERMEDIATE FORMS:




Evolving through stable intermediate forms to move from one state to another, allowing adjustment and adaptation en route, can be similar to a hill-climbing approach, where the overall goal is to reach the brow but each step is taken based on local terrain. However, with hill climbing it is possible to get caught in suboptimal local maxima. The INTERMEDIATE AND INCREMENTAL DEVELOPMENT of a system will ensure that its architecture addresses the needs to date, but it may find itself unable to meet a new need without significant reworking. STABLE INTERMEDIATE FORMS fortunately complements the risk of ascending local peaks blindly with the capacity to descend from them steadily. A clear vision not only of the ultimate goal but also of the general lay of the land ahead can help to mitigate the risk of getting into such situations – for example, a shared notion of a stable baseline architecture and the use of prototyping support architectural stability and evolution.

To stabilise an intermediate form is, in many respects, an act of completion. To put something into a finalised state, such as finishing a development increment, involves more effort in the short term than simply continuing without such a reflective and visible checkpoint. For development that is well-bounded in scope, occurs in a well-known domain, and is carried out with an experienced and successful team, a non-checkpointed strategy may sometimes be seen as appropriate, but otherwise measures must be taken to avoid letting other development variables and realities interfere, either by inserting STABLE INTERMEDIATE FORMS or by shoring up other variables through other guidance and feedback mechanisms, whether formal or informal.

STABLE INTERMEDIATE FORMS offers a path for change that offers similar consequences for open-ended and continued development as well as for more carefully scoped and goal-driven objectives. This applies as much to software development, for example the evolution of typical Open Source Software [Raymond2000], as it does to other domains, such as building construction [Brand1994].

In moving from a state that is inherently complete to another state there is also the implication that there is some rework involved, so that not all of the effort invested will be used in making externally visible progress, the remainder being used in revision or considered redundant. In some cases this can represent overhead and overcautiousness, particularly where the cost-benefit of a more

reticent and punctuated approach is not immediately clear. In others the return on investment in effort comes over time. Revision is not always an overhead: sometimes it is a necessary part of the creative process. The body renews itself over time to ensure both growth and the removal of damage. In STABLE INTERMEDIATE FORMS, stepping from one state to another allows consolidation as well as a simple change of state. In refactoring the same process of removing development friction over time can be seen [Fowler1999]:

> refactoring (noun): *a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*
>
> refactor (verb): *to restructure software by applying a series of refactorings without changing the observable behavior of the software.*

To clarify, it is the functional behaviour that remains unchanged with respect to a refactoring: the operational behaviour, e.g. performance or resource usage, may well change. Refactoring needs to be informed by other practices that provide confidence that a change is indeed stable and defect free, e.g. refactoring as an integral practice in Test-Driven Development (TDD) [Beck2003] or as complemented by some form of code review, whether a simple desk check with a colleague or a full walkthrough in a larger meeting.

Considering a test-driven microprocess as a variant of ITERATIVE AND INCREMENTAL DEVELOPMENT raises the question of step size. How large should the span between one intermediate form and another be? In the case of TDD the granularity of change is visible to an individual developer over a minute-to-hour timeframe, for continuous integration stable changes are visible to a development team over an hour-to-day timeframe, and for NAMED STABLE BASES [Coplien+2005] in an ITERATIVE AND INCREMENTAL DEVELOPMENT the visibility of change to all stakeholders occurs on the order of a day-to-week timeframe.

Class invariants present a non-process example of STABLE INTERMEDIATE FORMS where the process of change is taken over an object's lifetime and the step size is defined in terms of public methods on an object [Meyer1997]:

> *An invariant for a class C is a set of assertions that every instance of C will satisfy at all "stable" times. Stable times are those in which the instance is in an observable state: On instance creation... [and] before and after every [external] call... to a routine....*

A method may be too fine grained to establish a simple invariant simply [Henney2003], in which case a larger step size is needed. Granular methods that are used together can be merged into a COMBINED METHOD [Henney2000b] so that the footfalls between observable stable forms are further apart.

In situations where there is no need for revision, the balancing efforts required to support STABLE INTERMEDIATE FORMS may seem superfluous. Repeatedly achieving temporary stability may seem to incur additional effort and redundancy that has reduced return on investment. The context determines whether such an approach is genuinely wasteful or not. For example, in COPY BEFORE RELEASE a replacement representation object is always created and the current one discarded even in the event of self assignment. Although the self assignment is safe, the extra allocation-deallocation cycle might be seen as overhead. However, self assignment is not a typical mode of use, so this cost tends toward zero.

*Kevlin Henney*
kevlin@curbralan.com

## Acknowledgments

I would like to thank James Noble for his patience and shepherding of this paper for EuroPLoP 2004, Klaus Marquardt for being a party to this and a foil for the humour and foot dragging of both shepherd and sheep, and the members of the EuroPLoP workshop for their feedback: Paris Avgeriou, Frank Buschmann, Kasper von Gunten, Arno Haase, Wolfgang Herzner, Asa MacWilliams, Juha Pärssinen, Symeon Retalis, Andreas Rüping, Aimilia Tzanavari, and Dimitrios Vogiatzis.

The first notes on this pattern were scribbled during a presentation at a patterns seminar in 1997. I would like to thank Alan O'Callaghan for the session that provided the inspiration, but I would also like to apologise for then paying less attention to his slides than I did to my scribbles.

## References

[Brand1994] Stewart Brand, *How Buildings Learn*, Phoenix, 1994.

[Booch1994] Grady Booch, *Object-Oriented Analysis and Design with Applications, 2nd edition*, Addison-Wesley, 1994.

[Coplien1992] James O Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.

[Coplien+2005] James O Coplien and Neil Harrison, *Organizational Patterns of Agile Software Development*, Prentice Hall, 2005.

[Fowler1999] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Gibson2000] William Gibson, *All Tomorrow's Parties*, Penguin Books, 2000.

[Henney1997] Kevlin Henney, "Self Assignment? No Problem!", *Overload 20*, June 1997.

[Henney1998] Kevlin Henney, "Creating Stable Assignments", *C++ Report 10(6)*, June 1998, http://www.curbralan.com

[Henney2000a] Kevlin Henney, "C++ Patterns: Executing Around Sequences", *EuroPLoP 2000*, July 2000, http://www.curbralan.com

[Henney2000b] Kevlin Henney, "A Tale of Two Patterns", *Java Report 5(12)*, December 2000, http://www.curbralan.com

[Henney2001] Kevlin Henney, "Another Tale of Two Patterns", *Java Report 6(3)*, March 2001, http://www.curbralan.com

[Hoare1980] Charles Anthony Richard Hoare, "The Emperor's Old Clothes", *Turing Award Lecture*, 1980.

[Meyer1997] Bertrand Meyer, *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997.

[Raymond2000] Eric S Raymond, *The Cathedral and the Bazaar*, O'Reilly, 2000, http://www.tuxedo.org/~esr/writings/cathedral-bazaar/

[Sutter2000] Herb Sutter, *Exceptional C++*, Addison-Wesley, 2000.

[Wikipedia] http://en.wikipedia.org/wiki/Satisficing, definition of satisficing, June 2000.

*The original version of this paper was accepted for the EuroPLoP 2004 conference. The current version incorporates feedback from the conference workshop on the paper, as well as other revisions. The Overload version has only minor editorial and typographical differences from the version submitted for the final conference proceedings.*

# A Pair Programming Experience

**by Randall W. Jensen, Ph.D.**

Agile methods and extreme programming have risen to the forefront of software management and development interest over the last few years. Two definitions of *agile* are: (1) able to move quickly and easily, and (2) mentally alert. Both definitions rely on the capabilities of the people within the development process. The "Agile Manifesto" [1] published in Software Development in 2001 created a new wave of interest in the agile philosophy and re-emphasized the importance of people. One of the points highlighted in the Manifesto is "We value individuals and interactions over processes and tools." That does not mean processes and tools are evil. It implies the individuals and interactions (people) are of *higher priority* than processes and tools. Textbooks [2,3] have been written to describe the importance of people in these new software development approaches that have demonstrated improved productivity and product quality. "Extreme programming" [4] is one member covered by the umbrella of agile methods. "Pair programming" [5] is a major practice [6] of extreme programming.

The official definition of pair programming is two programmers working together, side by side, at one computer collaborating on the same, analysis, design, implementation and test. In other words – two programmers, one pencil.

We have all experienced elements of the pair programming concept in one way or another during our lives. How many times have we been stuck removing an error from a design or program with no success? When everything else failed, we went to our neighbour programmer, the "casual observer", to see if we could get some assistance. While explaining our problem, we have a flash of inspiration, and the problem is quickly solved. How much time did we waste before asking a neighbour for insight? Can we relate this to pair programming?

I was introduced to pair programming indirectly as an undergraduate electrical engineering student in the 1950s. The class and laboratory workload was such that any free time during the 4-year program was more wishful thinking than reality. Working part time made the program even more daunting. Fortunately, two other EE students in the same academic program were struggling with different sets of outside commitments. We decided to work together on homework assignments, lab work and test preparation to lighten the course load. We successfully maintained that approach through the entire program in spite of having been conditioned throughout our lives to perform solitary work. Our educational system does not condone or encourage teamwork. That education philosophy supports individual student evaluation, but works against learning. The teamwork concept became ingrained in my thinking, as well as in my programming and management research activities.

Later, much later, I was asked to find ways to improve programmer productivity in a large software organization. The undergraduate experience led me to propose an experiment in the application of what we called "two-person programming teams." The term pair programming hadn't been coined at that time. The experiment results are the subject of this case study.

## Development Task

### Problem

A description of the results achieved through the use of pair programming without knowledge of the project or development

---

# C Abuse

**by Thaddaeus Frogley**

*[Editorial note: I have often stated that one must consider the intended audience in any piece of writing. I've also been known to point out that code is a form of writing and that the principal audience, in this case, is not the compiler. This article demonstrates just how much the style of code can be affected by inverting the usual assumption that the intent is to communicate what the code does to your audience. - Alan]*

Four years ago I collaborated on the THADGAVIN [1] entry to the International Obfuscated C Code Contest [2]. It won an award (Most Portable Output), but it was a behemoth, only just scraping inside the maximum size limit for the contest. Much of its complexity was inherent in the algorithm, the obfuscation was a straightforward refactoring, and the reason for the award was mostly due to the use of a cross platform library. In conclusion, while it was a winner, in the end I was not happy with the "art" of the code.

The following year I wrote another entry. This time I wanted to do something functional and minimalist, something with a strong theme, and something self-referential. I had striven, and in my opinion failed, to make the 2000 entry a visual as well as an intellectual appeal. I wanted to do the same again in 2001, but again I wanted to be minimalist. I wanted to create a piece of code that could be considered to be "art" on multiple levels. So here, for your pleasure, I present my short (untitled) program:

```
/*(c) 2001 Thad */
#include<string.h>
#include <stdio.h>
#define abc stdout
int main(int a,ch\
ar*b){char*c="??="
"??(??/??/??)??'{"
"??!??>??-";while(
!((a=fgetc(stdin))
==EOF))fputc((b=s\
trchr(c,a))?fputc(
fputc(077,abc),abc
),"=(/)'<!>""-"??(
b-c??):a, abc);??>
```

I encourage you to study it. Try to work out what it does without compiling and running it.

Now, assuming you've worked out what it does, can you work out how many sins are committed in these 14 short lines of code? How many good practice guidelines are broken?

*Thaddaeus Frogley*
codemonkey_uk@mac.com

## References

[1] http://thad.notagoth.org/thadgavin/
[2] http://www.ioccc.org/
[3] http://developer.apple.com/documentation/DeveloperTools/gcc-3.3/cpp/Initial-processing.html

task underlying the experience would be meaningless. The software to be developed in this project was a multitasking real-time system executive. The product consisted of six independent components containing a total of approximately 50,000 source lines of code. The product contained no reused or COTS [“Commercial, Off-the-Shelf”] components. FORTRAN was the required software development language. The real-time executive was to be used to support the development of a large, complex software system by the developing organization. The development schedule for the executive was critical and short.

## Team Composition

The development team consisted of ten programmers with a wide range of experience and a manager. I tend to divide managers into two primary groups: Theory X [7,8] and Theory Y. The manager for this task was experienced and from the Theory Y group.

The ten programmers assigned to the executive development had prior experience that ran the gamut from an expert system programmer to a couple of fresh, young college graduates. None of these programmers had any experience working in a team environment. As a collection, I would place them as about average for that development organization.

The manager grouped the programmers into five teams according to their experience level. Each team pair was composed of the most experienced and least experienced programmer of the remaining group. The first team consisted of the expert system programmer and a person who had just returned from a six-year leave of absence. The fifth team consisted of two programmers of near equal capability and experience. These first and fifth programming teams were important in the way they impacted the project. I will address their impacts in the Lessons Learned.

No special changes from normal were made to the development environment. The facilities were essentially 2-person cubicles. The programming pairs were co-located in these cubicles. Each cubicle contained two computer workstations, two desks and a common worktable. The pair programming approach dictated that the pair (two programmers, one pencil) uses only one development terminal located on the common worktable. The second terminal was be used for documentation, etc. not related to the team’s assigned development.

One programmer of the pair functioned as the “driver” operating the keyboard and mouse, while the second programmer functioned more as a “navigator” or “co-pilot.” The navigator reviewed, in real-time, the information entered by the driver. The roles of the two programmers were not permanent; frequent role changes occurred daily. The navigator was not a passive role at any time.

## Results

### A Priori

A productivity and error baseline for the project could not be directly obtained for the project individuals, but data was available from past projects that allowed us to project productivity and error averages for the project. The average productivity and error rates in most organizations with consistent management style and processes are near constant and quite predictable. The baseline productivity was determined to be approximately 77 source lines per person-month. The error rate for the development organization was normal for the aerospace industry. The numerical error rate value is not significant for this presentation, and will remain unknown.

Formal design walkthroughs and software inspections were not scheduled for this project. The project would follow a classic waterfall development approach, which is inconsistent with today’s agile methods. Formal preliminary and critical design reviews, as well as a final qualification test, were planned. Formal review and test documentation was reduced to essential information; that is, all elements necessary to proceed with the development.

### A Posteriori

The productivity achieved in the real-time executive development was 175 source lines per person-month as shown in Table 1. We hoped for a productivity gain of anything greater than 0 percent. Any small gain would have compensated for the two programmers loading on each task. The 127 percent gain achieved was phenomenal and a cause for celebration.

The error analysis showed the project had achieved an error rate that was three orders of magnitude less than normal for the organization. Integration of the first two components (approximately 10,000 source lines) was completed with only two coding errors and one design error. The third component was integrated with no errors. The remaining three components had more errors, but the number of errors for these components was significantly less than normal.

| Topic | Historical | Pair Results | Gain |
|---|---|---|---|
| **Productivity (lines/person-month)** | 77 | 175 | 127% |
| **Error Rate** | - | - | 0.001 x normal |

**Table 1. Pair programming productivity and error rate gains**

The “continuous walkthrough” assumption was demonstrated to be very effective, and more than compensated for the lack of formal walkthroughs. The formal preliminary and critical design reviews, as well as a final qualification test, were effective in keeping the five teams coordinated. Few problems were uncovered in the review and test activities.

After the experiment was completed, the development manager presented the very positive results to the organization’s management staff. The project managers’ reaction to the results was memorable. They claimed that their senior programmers would quit before they would team with another programmer. The use of pair programmers was never implemented in that organization.

## Lessons Learned

Several positive and some negative characteristics were observed during the pair programming experiment. In general the attributes of the college experience were exhibited here. The positive attributes, not necessarily in any order, are as follows:

- **Brainstorming** – According to the programmers, active real-time produced higher quality designs than would have been achieved working alone. Little time was lost optimizing code with more than one brain working.
- **Continuous design walkthrough** – The design and code were reviewed in real-time by both programmers who ultimately

produced fewer errors in each team product. Classic walkthroughs and inspections are, whether we like it or not, somewhat adversarial. The continuous walkthroughs within the team were more positive and supportive.

- **Focused energy** – The individual teams appeared to be more focused in their activities. The highly visible aspect of this attribute was the programmers took fewer breaks for restrooms, coffee, outside discussions, etc.
- **Mentor** – When we started work in this industry, we were usually told about on-the-job training that never materialized. Pair programming, when the two programmers were not of the same experience level, provided a craftsman/apprentice relationship that elevated the junior programmer's skill quickly. Conversely, the craftsman's skill is extended by the apprentice's questions and thinking outside of the box.
- **Motivation** – In general, the programming pairs appeared much more motivated than their single counterparts. The motivation level cannot be solely attributed to the pair concept or the experiment itself. Some of the motivation must be attributed to the project manager. Some has to be attributed to rapid progress and the product quality. One of the Theory Y assumptions is that motivation occurs at the social, esteem and self-actualization levels, as well as physiological and security levels.
- **Problem isolation** – The time wasted with two pairs of eyes (or brains) is significantly less than the amount of time wasted trying to solve a problem in isolation.

The negative observations cannot be ignored. The important observations, not necessarily in order of importance, are as follows:

- **Programming pair of the same experience and capability level is often counter-productive.** The most troublesome pairs we dealt with during the experiment were two teams in which both members were near the same capability level. The worst case team consisted of two "prima donna" programmers. The programming pair theoretically has equal responsibility for the team's efforts and product. We found teams functioned more smoothly, in spite of the members equally being driver and navigator, if one member was slightly more capable than the other. I read a statement by a software industry leader that stated hiring software engineers from the top ten percentile of the top ten universities would produce the best software development teams. I cannot imagine the stress that many egos can create on one project. Two strong egos of any caliber on a team create chaos until they recognize the power of two minds.
- **Coordination between the five teams would have improved if the teams had been working in a common area.** Each team was located in a two-person cubicle, which limited the interaction between the teams. I use the term *war room* (or skunk works) to describe the ideal open environment, which would be a large area with worktables in the centre and cubicles around the outside.

Some additional characteristics of the successful experiment are worthy of note. First, one of the manager's principal responsibilities was to buffer the teams from outside interference. The manager listed other important responsibilities that included referee (in the case of the prima donnas), arbitrator, coordinator, planner, cheerleader, and supplier of popcorn and other junk food.

Second, project managers must be supportive of the pair programming process. A classic (Theory X) manager observed a programming pair working on a design over a period of time. This manager suggested to their supervisor that one of the two programmers be laid off because only one was doing anything constructive. (The driver always gets the credit.) When the supervisor heard the suggestion, he replied that these programmers were the most productive people in the organization. The manager answered that the programmers keep their office door closed so others would not get the same idea.

## Summary and Conclusions

Most managers who have not experienced pair programming reject the idea without trial for one of two reasons. First, the concept appears redundant and wasteful of computing resources. Why would I want to use two programmers to do the work that one can do? How can I justify a 100 percent increase in person-hours to use this development approach? The project cannot afford to waste limited resources.

The second reason is the assumption that programmers prefer to work in isolation. Programmers, like most other people, have been trained to work alone. Yet, according to the 1984 Coding War Games sponsored by the Atlantic Systems Guild, only one third of a programmer's time is spent is isolation; two-thirds of the time is spent communicating with team members. Managers wonder about the adjustments necessary to adjust to another's work habits and programming style. They also worry about ego issues and disagreements about the product's implementation.

This experiment demonstrated strongly that programmers can work together effectively and efficiently to produce a quality product of which both programmers can be proud. Prior programming experience is not an issue. There are situations that initially occur, especially with a team of equal experience and ego, where disagreements over who will be the driver arise. Those situations are generally transient. The benefits listed in the Results Section overwhelmed any personality issues that arose.

The second major benefit demonstrated in this experiment, a three order of magnitude improvement in error rate, is hard to ignore. Repairing defects after developments are much more expensive to fix than uncovering and fixing the defects where they occur. The benefits of developing and delivering a stable product faster, reducing maintenance costs, and gaining customer satisfaction certainly minimizes the risk of using pair programming teams.

*Randall W. Jensen*
Randall.Jensen@HILL.af.mil

*This article has been previously published (as follows) and is used by permission of STSC.*

Jensen, Randall W., "A Pair Programming Experience," *CrossTalk: A Journal of Defense Software Engineering*, (Hill AFB, UT: Software Technology Support Center), March 2003

## References

[1] "The Agile Manifesto," *Software Development*, Vol. 9, No. 8, August, 2001

[2] DeMarco, T. and T. Lister, *Peopleware*, (New York: Dorset House Publishers), 1977

[concluded at foot of next page]

# The Developer's New Work

**by Allan Kelly**

I read Stefan Heinzmann's piece(s) in Overload 61 with a feeling of déjà vu. Not, you understand, because I'd wrestled with the same coding problem as him – if you recall he just wanted to store look up tables in ROM. No, the feeling of déjà vu came from the other problem he was wrestling with: What is this monster we have created called C++?

For me this thought is quickly followed by: How can anyone ever expect to master it? And then: How can I expect anyone to ever maintain this code?

Now I always consider the Overload readership to be a pretty savvy bunch. People who are, on the whole, smarter than the average C++ developer, but how many of us could have tackled Stefan's little task without encountering many of the same problems? I'll go out on a limb, I don't think any Overload reader could have tackled that problem and got it right in one sitting. I'll go further, I don't think even Herb Sutter, Bjarne Stroustrup or Andrei Alexandrescu could have done it in one sitting.

Partly that's because Stefan was engaged in a learning exercise. As he developed a little code his ideas became more refined. The compiler forced him to remove every ambiguity from his original idea. Hence, when he found the compiler inadequate or vague he was lost.

The other part of his problem was that he was attempting to use the compiler and language to the full. This was where the real problems set in. This is where my sense of déjà vu came from.

## Blast From the Past

Once upon a time I wrote an application. I thought it was a good application, it was reliable (mostly), was fast, performed its job and was easily understood through a few design patterns and employed thoroughly modern C++ and standard library. Then it came time to leave the company.

The company wasn't a bad company so they selected a developer to take over my work. He'd been with the company a few years but had mostly done Visual Basic work. So they company sent him on a course to learn C++. And then I tried to hand over to him.

Out of that experience came a short little piece of angst entitled *High Church C++* – for reasons which I don't recall it never made it into the pages of Overload but has been a popular download from my web-site (Kelly, 2000). In *High Church C++* I wrestled with my conscience, was it right to write a program in a style which was endorsed by an elite but foreign to the masses?

In truth, it wasn't necessarily my style of writing, had I written "low church C++" or some "MFC C++" style the code would have been superficially clearer to the novice but the increased length would have added to the complexity. Complexity can be like that, you push it down in one place and it appears somewhere else.

The APPRENTICE pattern (Coplien and Harrison, 2004) suggests it can take a year for someone to become proficient in a new system. What is the best way to bring someone up to speed? Is High-Church C++ better than Low-Church C++? Is Java better than C++? What of Visual Basic? C#?

Whatever our language, whatever our choice of idioms, patterns and style, the same problem exists. Someone ends up wrestling with the code base to understand it.

## Déjà Vu All Over Again

Contemporary C++ with heavy use of templates, standard library and even exceptions adds a twist because it seems C++ has become two different languages. What I called High-Church C++ and Low-Church C++ might also be called Modern C++[1] and Classic C++. Whatever we call them there seems to be a disconnect between the two schools. (If I recall correctly Kevlin Henney describes *three ages of C++*.)

So it was I had the same déjà vu again a few weeks ago. My office book group has been looking at Herb Sutter's Exceptional C++ (Sutter, 1999). In our discussion on exception handling several people suggested that the nuances and intricacies of exception handling meant it was too complicated to use. I think they have a valid point, it is too complicated to use: the stack unwind, the need to avoid resources leaks, the care and attention needed to every line – for heavens sake, it took the brightest C++ minds nearly 5 years to answer Cargill's stack problem (Cargill, 1994).

But what's the alternative?

To my mind the alternative to C++ style exception handling is worse. It's so much worse in fact that people don't do it. They simply ignore error handling[2]. C++ style exception handling forces you to face up to resource management, error handling and the like, but it does it at a cost. Get it wrong and the result can be awful.

The alternative isn't the alternative people think it is. The alternative, let's call it "return codes," suffers from most of the same

1 I use the term *Modern C++* in a broader sense than Andrei Alexandrescu's book *Modern C++ Design*; that book is a good example of *Modern C++*.

2 For several years it was my point of view that the main difference between the programming we learnt in college and that we did in industry was *"Error handling omitted to save space"* – a popular textbook expression that doesn't cut it in industry.

---

[continued from previous page]


[3] Weinberg, G. M., *The Psychology of Computer Programming Silver Anniversary Edition*, ((New York: Dorset House Publishers), 1998

[4] Beck, K., *Extreme Programming Explained: Embracing Change*, (Reading, MA: Addison-Wesley),2000

[5] Williams, L., R. R. Kessler, W. Cunningham and R. Jeffries, "Strengthening the Case for Pair Programming," *IEEE Software*, Vol. 17, No. 4, (July/August 2000), pp. 19-25

[6] Beck, K., "Embracing Change with Extreme Programming," *Computer*, October, 1999, pg.71

[7] Hersey, P. and K. H. Blanchard, *Management of Organizational Behavior, Utilizing Human Resources*, (Englewood Cliffs, NJ: Prentice-Hall, Inc.), 1977

Theory X assumes: 1. Work is inherently distasteful to most people. 2. Most people are not ambitious, have little desire for responsibility, and prefer to be directed. 3. Most people have little capacity for creativity in solving organizational problems. 4. Most people must be closely controlled and often coerced to achieve organizational objectives.

Theory Y assumes: 1. Work is as natural as play, if conditions are favourable. 2. Self-control is often indispensable in achieving organization goals. 3. The capacity for creativity in solving organizational problems is widely distributed in the population. 4. People can be self-directed and creative at work if properly motivated.

[8] McGregor, D., *The Human Side of Enterprise*, (New York: McGraw-Hill Book Co.), 1960

problems but it's easier to hide from the problem and pretend you're doing it right.

It's the complexity problem again. We push down complexity in one place by establishing syntax and conventions in the language to support good error handling, but more complexity arises in getting developers to understand and use the conventions.

## C++ is Not Alone

I've been talking about C++ because that's where my personal experience is, that's where Stefan had his problem and that's what most Overload readers program in (I think.) But the problem is not confined to this.

Other languages and technologies have their own problems. Java too has exceptions – and imposes more complex syntax and idioms to handle them correctly than does C++, Perl has its "write only syntax", nor should we think only of programming languages, Unix, Linux and NT all have hidden depths.

And to make things worse we can't just specialise in C++ or Linux, we need to know a cross section: language, OS, database, development techniques. How can we ever keep up?

## Searching For a Solution

The angst in *High Church C++* was very real, I was scared. But what was the answer? I've been looking for a solution for five years now.

We could "dumb down" our code, make it really simple. Trouble is, we have real problems and we need real solutions. To tackle the same problem with "low Church code" just moves the complexity from the *context* to an overly verbose code base.

We could just hire real top-gun programmers. This isn't really a solution; once again we're pushing the problem down in one place and seeing it come up in another. Since there aren't that many super-programmers in the world finding them is a problem, keeping them a problem, motivating them is a problem and even if we overcome these problems it's quite likely that within our group of super-programmers we would see an elite group emerge.

Hiring a group of super-programmers is in itself an admission of defeat, we're saying: *We don't know how to create productive employees; we're going to poach people from companies who do*. In doing so we move the problem from our code to recruitment.

So, maybe the solution is to get management to invest more in training. But this isn't always the solution. The managers I had when I wrote *High Church C++* tried to do the right thing. Is it not reasonable to assume that someone who has been on a C++ course can maintain a system written in C++? As Alan Griffiths pointed out, this is about as reasonable as expecting someone that has been on a car maintenance course to change the tyres during an F1 pitstop.

The answer of course is: No, knowing C++ is a requirement for maintaining a C++ based system but it isn't sufficient of itself. One needs to understand the domain the system is in and the system architecture – this is why Coplien and Harrison say you need a whole year to come up to speed.

How do we communicate these things? The classical answer is "write it down" but written documentation has its own problems: accuracy, timeliness, readability, and memorability to name a few. In truth, understanding any modern software system is more about tacit knowledge than it is about explicit knowledge.

So what are we to do?

I don't claim I have the only answer, I don't claim I have the best answer, I don't claim my answer even covers all the bases and it certainly isn't original. But after five years of searching I think I have *an answer*, at least it's the best answer I know at the moment.

The answer to the question, the great question, the question of *how do you teach someone about a software system...* is... well, you aren't going to like it....

## New Work

The developer's new work is to help others learn. I don't mean you all rush off and become C++ trainers, I mean we all need to work to improve the capabilities of our colleagues and especially the less experienced around us. This isn't about training, it is about learning. It's about redefining what it means to be a software developer.

Implicit here is another role, to lead. When I mix with other ACCU members I find I share an unspoken bond with them. We all believe it is possible to write better software. Exactly how we do this may be up for debate, maybe we should adopt Extreme Programming, or maybe write in Java, or simply write our tests first. These are all good answers, the real question is: how do we get from here to there?

It is no longer enough to just cut code. Sure you may need to do this too, but if you want to use modern C++ (or modern Java, Python, or what ever) it is your job to lead others in a change. And change doesn't happen without learning. Indeed, learning isn't really happening if we don't change, we may be able to recite some piece of information but unless we act on it we haven't really learnt anything.

So, when it comes to improving your code it isn't enough to sit your colleagues down and tell them that a template-template function is the thing they need here and expect them to make it so. You've imparted information, you may even have ordered them to do it, but they haven't been led, they haven't learnt and they won't have changed – they'll do the same thing all over again.

Simply informing people "This is a better way" doesn't cut it. You can't lecture, you can't tell, you can't enforce conformance. You need to help others find their own way to learn. Helping them find that way goes beyond simply giving them the book, they need to be motivated, people who are told aren't motivated, people who are ordered aren't motivated; motivating people requires leadership.

If you find yourself resorting to a rational argument to persuade someone to do it your way you have failed. Your work is to help them produce the rational argument themselves. We aren't abandoning rationality, just recognising that when you tell someone "you are wrong, I am right" it doesn't do much, far better to help them realise a better way.

For many people simply being told "the correct way," a "better way" isn't enough to bring about a change in their actions. It may well demonstrate your intellectual superiority over them but it isn't going to change them. Telling them to "Read the frigging manual" or "Read my document" isn't useful.

Of course, this is easier said than done...

## What Do I Do Now?

You need to change your mindset, you are no longer out to prove you know C++ better than anyone else, you are here to lead them in learning. Because you know C++ better than anyone else you are in a position to do this.

The first change is to stop doing something: stop switching people off. Telling people they are wrong, fixing their problems – especially problems they don't realise are problems – is a sure fire

way of switching people off. Why learn something or do it the proper way if someone else will always do the job for you?

Next you need to create awareness of the problem. Are your developers really aware that there is a problem with SINGLETON pattern? Of course, you can't just tell them. Well, maybe you can tell them, but it needs to be in an abstract kind of way, a way that will spark their curiosity, help them find the problem themselves.

I can hear some people saying "But the developers I work with just don't care." These are developers who have been switched off. People are learning machines, if people have given up learning about these things then why? Maybe they've been punished in the past for free thinking. Maybe your company rewards conformance, rocking the boat isn't positive.

Of course, you don't want to be seen as a boat rocker do you? Maybe you do, maybe what you value is demonstrating how much better your insights are? If so your attitude hasn't changed. You want to be somewhere between sparking curiosity and enquiry, so, throw away those Dilbert cartoons.

You need to redefine your own job and your own self-image. Start with yourself, as you are the only person you have complete control over. What is stopping you helping others? Recognise the barriers and over come them. Now move on and do the same with your colleagues.

## But I Don't Have the Time...

None of us have enough time, but if you're spending your time rushing around fixing other people's problems and policing their actions you're going to have even less time. There is never time to start doing something new, so it is always the right time.

If you wait, the *right* time will never come along, there will never be a day when your project has finished and the new one hasn't begun. And should that day come it's probably a sign that you've done something wrong.

Sure it's hard to adopt a new way of working a few days before a project deadline so maybe this isn't actually the right time. But if you are starting a project, or you're half way through, and you can't find the time to change you never will.

As the people around you learn the new techniques and grow in confidence you will find you don't need to spend so much time fixing their work and fire fighting.

## Again, But What Do I Do Now?

I am sorry, dear reader, but I have to disappoint you. I have no list of 35 things to do, I have no "Effective Learning" to give you. You have to find your own "answers which work" for you, your team and your company.

Anyway, I'm not the best person to ask on this subject. More worthy authors than me have examined this question and they are the place to start. This isn't the first Overload article by me to cite Senge's *The Fifth Discipline* (1990) but I don't apologise for suggesting you read it[3].

In dealing with people we are letting go of the Swiss-army knife of rationality, emotions are more important so Goldman's *Emotional Intelligence* (1996) is well worth a read (and particularly so if you're a parent too.)

Somewhere along the line you should seek to improve yourself, although its not my favourite book and I don't agree with everything he says Covey's *Seven Habits of Effective People* (1992) is the book to set you thinking about yourself. For those who can't find the time at least absorb his fifth habit: *"Seek first to understand, Then to be understood."* (Covey also has a solution to the "don't rock the boat" problem but I'll let you read that for yourself.)

If you've made it this far your attitude will already be changing and you'll be looking for new ways of working. After all this reading you'll either be ready to dismiss my ideas or take up the challenge. The next two books are more practical. You might want to try coaching, Whitmore (2002) provides a good introduction to the subject. You'll also recognise the need to spread your ideas subtly, this is where Linda Rising's new book is useful. *Fearless Change* (Manns and Rising, 2005) is a recipe book for introducing new ideas. Pick a pattern try it, see what happens. Try another.

Maybe I'm ducking the issue. I haven't given you any hard and fast rules. I haven't said "Say X to your developers" but I don't think I really can. You have to first learn yourself. Sure I could give you a quick check-list of do's and don'ts but I do not know you, your developers or your environment. The books I've listed here won't give you all the answers but they should help you find your own answers.

## We're a Long Way From Where We Started...

There are no silver bullets. Technical solutions have a habit of becoming technical problems – like Stefan's templates. Or, to put it another way:

> *"Today's problems come from yesterday's 'solutions'." (Senge, 1990, p.57)*

Hope lies not in code, not in machines but in people. If we believe that Modern C++ is best – and I truly, rationally, believe it is – then I have no choice other than to develop the people around me – and that belief is rational too.

So, I'm not giving you any silver bullets but I am telling you where you can hunt for silver. True, it will take you time to find the silver and you'll need some help making the bullets so you have plenty of opportunity to practise leaning and leading.

*Allan Kelly*
`www.allankelly.net`

## Bibliography


Cargill, T. 1994, "Exception Handling: A False Sense of Security", *C++ Report* 6, http://www.awprofessional.com/content/images/020163371x/supplements/Exception_Handling_Article.html

Coplien, J. O. and Harrison, N. B. 2004, *Organizational Patterns of Agile Software Development*, Pearson Prentice Hall, Upper Saddle River, NJ.

Covey, S. R. 1992, *The Seven Habits of Highly Effective People : Restoring the Character Ethic*, Simon & Schuster, London.

Goldman, D. 1996, *Emotional Intelligence*, Bloomsbury.

Kelly, A. 2000, *High Church C++*, http://www.allankelly.net/writing/WebOnly/HighChurch.htm

Manns, M. K. and Rising, L. 2005, *Fearless Change – Patterns for Introducing New Ideas*, Addison Wesley, Boston, MA.

Senge, P. 1990, *The Fifth Discipline*, Random House Books.

Sutter, H. 1999, *Exceptional C++*, Addison-Wesley.

Whitmore, J. 2002, *Coaching for Performance GROWing People, Performance and Purpose*, Nicholas Brealey, London.


1 Much of this essay is inspired by Senge's chapter entitled *"The Leader's New Work."*

# C++ Properties – a Library Solution

by Lois Goldthwaite

Properties are a feature of a number of programming languages – Visual Basic and C# are two of them. While they are not part of standard C++, they have been implemented in C++ for the CLI (popularly known as .Net) environment and Borland C++ Builder, which comes with a library of useful components which make heavy use of properties in their interfaces. And when programmers are asked their ideas for extensions to C++, properties are a popular request. [1,2]

So what is a property? The C++/CLI draft spec says, *"A property is a member that behaves as if it were a field... Properties are used to implement data hiding within the class itself."* In the class declaration, properties must be declared with a special keyword so the compiler knows to add appropriate magic:

```
class point {
private:
  int Xor;
  int Yor;
public:
  property int X {
    int get() {
      return Xor;
    }
    void set(int value) {
      Xor = value;
    }
  }
  ...
};

point p; p.X = 25;
```

In application code, properties look like ordinary data members of a class; their value can be read from and assigned to with a simple = sign. But in fact they are only pseudo-data members. Any such reference implicitly invokes an accessor function call "under the covers" and certain language features don't work with properties. One example is taking the address of a data member and assigning it to a pointer-to-member.

Let me say up front that I dislike this style of programming with properties. The subtle advantage of "hiding" a private data member by treating it as a public data member escapes me. And from the object-oriented-design point of view, I think they are highly suspect. One of the fundamental principles of OO design is that behaviour should be visible; state should not. To conceal behaviour by masquerading it as a data member is just encouraging bad habits.

Properties are syntactic saccharine for getter and setter member functions for individual fields in a class. Of course, in the quest for good design, replacing a public data member with simple get/set functions for that data member does not achieve a large increase in abstraction. A function-call syntax which does not overtly refer to manipulating data members may lead to a cleaner interface. It allows the possibility that some "data members" may be computed at runtime rather than stored. An overwhelming majority of books on OO design recommend thinking in terms of objects' behaviour, while hiding their state. Let us encourage good habits, not bad ones.

UK C++ panel member Alisdair Meredith, with almost a decade's daily experience using the Borland VCL properties, had these comments:

> *Properties work fantastically well in RAD development where you want interactive tools beyond a simple source code editor. For all they appear glorified get/set syntax, they make the life of the component writer much simpler. There are no strange coding conventions to follow so that things magically work, and a lot of boilerplate code vanishes. From this perspective they are most definitely A Good Thing™.*
>
> *Of course the property concept goes way beyond simply supporting GUI tools, and that is where the slippery slope begins...*
>
> *If functional programming is 'programming without side effects', property oriented programming is the other extreme. Everything relies on side effects that occur as you update state. For example: you have a `Left` property in your GUI editor to determine position of a control. How would you move this control at runtime? Traditionally we might write some kind of `Move()` function, but now we can set the`Left` property instead and that will somehow magically move the control on the form as a side-effect, and maybe trigger other events with callbacks into user code as a consequence.*
>
> *Experience shows that people prefer to update the `Left` property rather than look for some other function. After all, that is how you would perform the same task at design/compile time. Generally, code becomes manipulating properties (and expecting the side effects) rather than making explicit function calls. Again, this is the 'minimum interface' principle so that there is one and only one simple way of achieving a given result. Typically, the `Move()` function is never written, and this reinforces the programming style.*
>
> *As we move beyond the GUI-tools arena, I find the property syntax becomes more and more confusing. I can no longer know by simple inspection of a function implementation if I can take the address of everything used as a variable, or even pass them by reference. This only gets worse when templates enter the mix.*

And the problem becomes worse yet when developers become fond of using write-only properties – values which can be set to achieve the side effect, but can never be queried.

The one-sentence summary of his experience is that using properties in non-GUI classes did not help productivity: *"Properties made our code easier to write, but immensely more difficult to maintain."* My own experience in trying to debug code with properties bears this out. While stepping through some code to look for leaks of memory and COM interfaces, I was examining all variables by hovering the mouse over them. Multiple hovers over the same variable (or what appeared to be a simple variable) showed up a data structure with different values each time, because the underlying property function was creating an additional COM interface with each access.

My impression is that the main benefit envisioned for properties is not their syntactic sleight-of-hand but (1) their ability to be discovered through introspection/reflection and manipulated non-programmatically by some sort of Rapid Application Development tool, and (2) their ability to be stored (I think "pickled" is the term used with Java Beans) in this configured state, so that they can be loaded at runtime, already configured.

[1] appears to take for granted that these features should come as a package, if standard C++ were to be extended to embrace properties.

However I might feel about using properties, I have to recognise that many people do find them useful, at least for certain types of applications. For people who do like this style of programming, at least some of the benefits can be achieved through library classes without changing the C++ language and compilers. The accompanying sample code defines some utility class templates which may be used as members of domain classes:

**Property** – a read-write property with data store and automatically generated get/set functions. This is what C++/CLI calls a trivial scalar property.

**ROProperty** – a read-only property calling a user-defined getter.

**WOProperty** – a write-only property calling a user-defined setter.

**RWProperty** – a read-write property which invokes user-defined functions.

**IndexedProperty** – a read-write named property with indexed access to own data store.

For programmer convenience these classes offer three redundant syntaxes for accessing their data members (or apparent data members – properties need not have real storage behind them):

- function call syntax
- `get`/`set` functions
- `operator = (T)` and `operator T()` for assignment to and from properties.

The read-only and write-only property classes implement only the accessors appropriate to their semantics, but for compatibility with C++/CLI they do reserve (unimplemented) the unnecessary `get` or `set` identifier.

Instances of the utility templates as outlined in this paper do have an address, they have a type for template deduction, and they can be used in a chain of assignments. They can also be used with today's compilers and debuggers. If someone brings forward a RAD tool to read and manipulate properties through metadata, then a "property" modifier or some other marker for the tool could be added to their declaration to help in generating the metadata.

This technique, described in its basic form in C++ Report back in 1995[3], seems at least as convenient, and produces as economical source code, as most uses of properties. The basic assignment and return functions can be inlined, and therefore should be efficient. In order to support chaining, the setter functions return their new value, but for complete compatibility with C++/CLI they should have a void return type.

One objection to these that has been raised is that CLI properties allegedly take up no space inside a class, whereas a C++ subobject cannot have a size of 0 (ignoring certain clever optimizations). On the other hand, I expect that most useful properties will need some way to preserve state between `set` and `get`, and that has to take up space somewhere. The size of `Property<T>` takes up as much space as a single object of its template parameter, while the other three hold only a single pointer to an implementation object. Note that the `Object` and member function template parameters do not have to refer to the containing object, though that is likely to be the most common usage. They could delegate the processing to an external or nested helper class. The choice of implementation object (though not its type or member functions) can even be changed dynamically, and several objects could share a single implementation object.

The biggest inconvenience to these classes that I perceive is that all but the simplest `Property<T>` instances need to be initialized at runtime with the address of their implementation object (usually the containing class). A smaller inconvenience is that using them on the right hand side of an assignment invokes a user-defined conversion, which could affect overload resolution and a sequence of conversions.

One of the features of C++/CLI properties is that they can be declared as virtual or pure virtual, for polymorphic overriding by derived classes. This obviously is not possible with data member declarations. But if the implementation object (which usually means the containing object) is itself polymorphic and the designated member functions are virtual, then the property will behave in a polymorphic fashion.

The C++/CLI feature of default (nameless) indexed properties can be simulated with a member `operator []` (except that in C++/CLI indexes can take multiple arguments, but there is a separate proposal for C++ to allow comma-separated arguments inside square brackets).

Aside from matters of stylistic preference, I can see two possible scenarios in which these classes might be useful. One scenario would be in source code which needs to be portable between C++/CLI and standard C++ environments. Using the `Property<T>` templates on the standard C++ side could compensate for a lack of compiler support for properties.

The second scenario would be for migrating code which uses public data members to a more encapsulated style. The class definitions could be rewritten to use the `Property<T>` templates, while client code need not be rewritten, only recompiled.

A brief sample program (at the end of this article) shows the utility classes in use. As it illustrates three different styles of setting and getting a property value, the expected output is this:

```
Name = Pinkie Platypus
Name = Kuddly Koala
Name = Willie Wombat

ID = 12345678
ID = 9999
ID = 42

Children = 42
Children = 42
Children = 42

WO function myClass::addWeight called with
                              value 2.71828
WO function myClass::addWeight called with
                              value 3.14159
WO function myClass::addWeight called with
                              value 1.61803
```

```
Secretkey = *****
Secretkey = !!!!!
Secretkey = ?????

Convenor = Herb
Minutes = Daveed
Coffee = Francis
```

*Lois Goldthwaite*

## References


[1] John Wiegley, *PME: Properties, Methods, and Events.* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1384.pdf

[2] David Vandevoorde, *C++/CLI Properties.* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1600.html

[3] Unfortunately I don't have a more exact reference to hand.


## Initialisation Methods for Properties

Overload Technical Reviewer Phil Bass raised this question while reviewing the article for publication:

> *The ROProperty, WOProperty and RWProperty classes provide a function call operator for initialisation. Why don't they have constructors instead?*

After looking over the code now (it was originally written last April) I have added a default constructor which makes sure the pointers are set to `0` for safety. But they still need to have an initialisation function. Initialising them, either with a named function or an overloaded function call operator, gives runtime flexibility. These are the benefits that occur to me:

- If the implementation object were initialised in a constructor, it would create a dependency on the prior existence of the implementation object. In a complex application where such dependencies might be chained or even circular, this would be a nuisance.

- The implementation object can be rebound. I don't know of any specific use cases that would drive this, but the capability falls out of doing it this way and might prove useful.

- One implementation object could serve several different instances of objects with a property. Again, it just falls out. If pressed to come up with a use case, I would look for a situation where the 'property' has to be extracted from some heavyweight situation – a database, a large table in memory, a remote procedure call, or some kind of bottleneck where localised optimisations like caching could make a difference. This encapsulates the resource issues and shares the benefits around.

- Since the function call operator (or some initialisation function) has to exist anyway, adding a constructor call which takes a value might confuse some programmers as to which they were calling. It wouldn't confuse the compiler – `NumberOfChildren(p)` in an initialiser list for `myClass` would invoke the constructor, whereas `NumberOfChildren(this)` inside a `myClass` constructor is calling the `operator()()` – but they look alike in code and many people are kind of vague about such details (which is a reason why many people avoid function objects).

- The combination of the first and last points above brings us to the REAL reason why we don't initialise it in a constructor: the most common use case is likely to be that the enclosing object exposing the property provides its own implementation for it. If we need to initialise a data member with a constructor, we would naturally expect to do so in the initialiser list of the enclosing object's constructor, like this:

```
myClass() : NumberOfChildren(this),
            WeightedValue(this),
            Secretkey(this) {}
```

But as the `myClass` instance doesn't really exist before its constructor is entered, using its this pointer as an argument to the member constructors in the initialiser list seems, well, dubious. Of my three test compilers only MSVC++ gave a warning that "the this pointer is only valid within nonstatic member functions." All three compiled and ran this program, but I wouldn't want to rely on that as proof of correctness. I'm afraid I did not devise a stress test involving a polymorphic hierarchy to see how far I could push it before something broke.

I did think hard about what syntax in a property class constructor would be able to deduce the address of the property's enclosing object as a default – that would be really useful! – but concluded there is no way to say that in C++. So, since the user of the class will have to initialise it at some point, not having a constructor means that initialisation has to go into the body of the enclosing object's constructor rather than its initialiser list. I opted for the function call syntax as more concise than a named member function.

I admit that all this was written as a proof-of-concept rather than well-tested production code. For industrial-strength use, one needs to envision pathological scenarios that might arise. If the overloaded setter using function-call syntax ever takes a parameter of the same type as the implementation object (value has type `Object *`), there is going to be an ambiguity with the initialiser; for this situation the solution is a named initialiser member function and/or a named `set()` function. Another low-probability scenario (which means one that is guaranteed to bite you sooner or later) is using an external implementation object which is itself a `const` instance. The workaround for this would be another, slightly different, utility class template wrapping a `const Object * my_object;`

One of the advantages of implementing properties as library classes instead of a built-in language feature is that the programmer can vary or extend them as needed!

```
// Some utility templates for emulating
// properties — preferring a library solution
// to a new language feature
// Each property has three sets of redundant
// acccessors:
// 1. function call syntax
// 2. get() and set() functions
// 3. overloaded operator =

// a read-write property with data store and
// automatically generated get/set functions.
// this is what C++/CLI calls a trivial scalar
// property
template <class T>
class Property {
  T data;
public:

  // access with function call syntax
  Property() : data() { }
  T operator()() const {
    return data;
  }
  T operator()(T const & value) {
    data = value;
    return data;
  }

  // access with get()/set() syntax
  T get() const {
    return data;
  }
  T set(T const & value) {
    data = value;
    return data;
  }

  // access with '=' sign
  // in an industrial-strength library,
  // specializations for appropriate types
  // might choose to add combined operators
  // like +=, etc.
  operator T() const {
    return data;
  }
  T operator=(T const & value) {
    data = value;
    return data;
  }
  typedef T value_type;
            // might be useful for template
            // deductions
};

// a read-only property calling a
// user-defined getter
template <typename T, typename Object,
          T (Object::*real_getter)()>
class ROProperty {
  Object * my_object;
public:
  ROProperty() : my_object(0) {}
  ROProperty(Object * me = 0)
                : my_object(me) {}

  // this function must be called by the
  // containing class, normally in a
  // constructor, to initialize the
  // ROProperty so it knows where its
  // real implementation code can be
  // found.
  // obj is usually the containing
  // class, but need not be; it could be a
  // special implementation object.
  void operator()(Object * obj) {
    my_object = obj;
  }

  // function call syntax
  T operator()() const {
    return (my_object->*real_getter)();
  }

  // get/set syntax
  T get() const {
    return (my_object->*real_getter)();
  }
  void set(T const & value);
            // reserved but not implemented,
            // per C++/CLI

  // use on rhs of '='
  operator T() const {
    return (my_object->*real_getter)();
  }

  typedef T value_type;
            // might be useful for template
            // deductions
};

// a write-only property calling a
// user-defined setter
template <class T, class Object,
          T (Object::*real_setter)(T const &)>
class WOProperty {
  Object * my_object;
public:
  WOProperty() : my_object(0) {}
  WOProperty(Object * me = 0)
                : my_object(me) {}

  // this function must be called by the
  // containing class, normally in a
  // constructor, to initialize the
  // WOProperty so it knows where its real
  // implementation code can be found
  void operator()(Object * obj) {
    my_object = obj;
  }
```

```
  // function call syntax
  T operator()(T const & value) {
    return (my_object->*real_setter)(value);
  }
  // get/set syntax
  T get() const;
            // reserved but not implemented,
            // per C++/CLI
  T set(T const & value) {
    return (my_object->*real_setter)(value);
  }

  // access with '=' sign
  T operator=(T const & value) {
    return (my_object->*real_setter)(value);
  }

  typedef T value_type;
            // might be useful for template
            // deductions
};

// a read-write property which invokes
// user-defined functions
template <class T,
          class Object,
          T (Object::*real_getter)(),
          T (Object::*real_setter)(T const &)>
class RWProperty {
  Object * my_object;
public:
  RWProperty() : my_object(0) {}
  RWProperty(Object * me = 0)
               : my_object(me) {}

  // this function must be called by the
  // containing class, normally in a
  // constructor, to initialize the
  // ROProperty so it knows where its
  // real implementation code can be
  // found
  void operator()(Object * obj) {
    my_object = obj;
  }

  // function call syntax
  T operator()() const {
    return (my_object->*real_getter)();
  }
  T operator()(T const & value) {
    return (my_object->*real_setter)(value);
  }

  // get/set syntax
  T get() const {
    return (my_object->*real_getter)();
  }
  T set(T const & value) {
    return (my_object->*real_setter)(value);
  }
  // access with '=' sign
  operator T() const {
    return (my_object->*real_getter)();
  }
  T operator=(T const & value) {
    return (my_object->*real_setter)(value);
  }

  typedef T value_type;
            // might be useful for template
            // deductions
};

// a read/write property providing indexed
// access.
// this class simply encapsulates a std::map
// and changes its interface to functions
// consistent with the other property<>
// classes.
// note that the interface combines certain
// limitations of std::map with
// some others from indexed properties as
// I understand them.
// an example of the first is that
// operator[] on a map will insert a
// key/value pair if it isn't already there.
// A consequence of this is that it can't
// be a const member function (and therefore
// you cannot access a const map using
// operator [].)
// an example of the second is that indexed
// properties do not appear to have any
// facility for erasing key/value pairs
// from the container.
// C++/CLI properties can have
// multi-dimensional indexes: prop[2,3].
// This is not allowed by the current rules
// of standard C++
#include <map>
template <class Key,
          class T,
          class Compare = std::less<Key>,
          class Allocator
               = std::allocator<std::pair<
                            const Key, T> > >
class IndexedProperty {
  std::map<Key, T, Compare,
           Allocator> data;
  typedef typename std::map<Key, T, Compare,
                        Allocator>::iterator
          map_iterator;
public:

  // function call syntax
  T operator()(Key const & key) {
    std::pair<map_iterator, bool> result;
    result
      = data.insert(std::make_pair(key, T()));
    return (*result.first).second;
  }
```

```
  T operator()(Key const & key,
             T const & t) {
    std::pair<map_iterator, bool> result;
    result
      = data.insert(std::make_pair(key, t));
    return (*result.first).second;
  }

  // get/set syntax
  T get_Item(Key const & key) {
    std::pair<map_iterator, bool> result;
    result
      = data.insert(std::make_pair(key, T()));
    return (*result.first).second;
  }
  T set_Item(Key const & key,
            T const & t) {
    std::pair<map_iterator, bool> result;
    result
      = data.insert(std::make_pair(key, t));
    return (*result.first).second;
  }

  // operator [] syntax
  T& operator[](Key const & key) {
    return (*((data.insert(make_pair(
                  key, T()))).first)).second;
  }
};


// ================================
// and this shows how Properties are
// accessed:
// ================================

#include <string>
#include <iostream>

class myClass {
private:
  Property<std::string> secretkey_;

  // --user-defined implementation functions--
  // in order to use these as parameters,
  // the compiler needs to see them
  // before they are used as template
  // arguments. It is possible to get rid
  // of this order dependency by writing
  // the templates with slight
  // differences, but then the program
  // must initialize them with the
  // function addresses at run time.

  // myKids is the real get function
  // supporting NumberOfChildren
  // property
  int myKids() {
    return 42;
  }
  // addWeight is the real set function
  // supporting WeightedValue property
  float addWeight(float const & value) {
    std::cout << "WO function "
              << "myClass::addWeight "
              << "called with value "
              << value
              << std::endl;
    return value;
  }
  // setSecretkey and getSecretkey support
  // the Secretkey property
  std::string setSecretkey(
                    const std::string& key) {

    // extra processing steps here

    return secretkey_(key);
  }
  std::string getSecretkey() {

    // extra processing steps here

    return secretkey_();
  }

public:
  // Name and ID are read-write properties
  // with automatic data store
  Property<std::string> Name;
  Property<long> ID;

  // Number_of_children is a read-only
  // property
  ROProperty<int, myClass,
           &myClass::myKids> NumberOfChildren;

  // WeightedValue is a write-only
  // property
  WOProperty<float, myClass,
           &myClass::addWeight> WeightedValue;

  // Secretkey is a read-write property
  // calling user-defined functions
  RWProperty<std::string, myClass,
           &myClass::getSecretkey,
           &myClass::setSecretkey> Secretkey;

  IndexedProperty<std::string,
                std::string> Assignments;

  // constructor for this myClass object
  // must notify member properties
  // what object they belong to
  myClass() {
    NumberOfChildren(this);
    WeightedValue(this);
    Secretkey(this);
  }
};
```

```
int main() {
  myClass thing;

  // Property<> members can be accessed
  // with function syntax ...
  thing.Name("Pinkie Platypus");
  std::string s1 = thing.Name();
  std::cout << "Name = "
            << s1
            << std::endl;

  // ... or with set/get syntax ...
  thing.Name.set("Kuddly Koala");
  s1 = thing.Name.get();
  std::cout << "Name = "
            << s1
            << std::endl;

  // ... or with the assignment operator
  thing.Name = "Willie Wombat";
  s1 = thing.Name;
  std::cout << "Name = "
            << s1
            << std::endl;
  std::cout << std::endl;

  // The same applies to Property<> members
  // wrapping different data types
  thing.ID(12345678);
  long id = thing.ID();
  std::cout << "ID = "
            << id
            << std::endl;

  thing.ID.set(9999);
  id = thing.ID.get();
  std::cout << "ID = "
            << id
            << std::endl;

  thing.ID = 42;
  id = thing.ID;
  std::cout << "ID = "
            << id
            << std::endl;
  std::cout << std::endl;

  // And to ROProperty<> members
  int brats = thing.NumberOfChildren();
  std::cout << "Children = "
            << brats
            << std::endl;

  brats = thing.NumberOfChildren.get();
  std::cout << "Children = "
            << brats
            << std::endl;

  brats = thing.NumberOfChildren;
  std::cout << "Children = "
            << brats
            << std::endl;
  std::cout << std::endl;

  // And WOProperty<> members
  thing.WeightedValue(2.71828);

  thing.WeightedValue.set(3.14159);

  thing.WeightedValue = 1.618034;
  std::cout << std::endl;

  // and RWProperty<> members
  thing.Secretkey("*****");
  std::string key = thing.Secretkey();
  std::cout << "Secretkey = "
            << key
            << std::endl;

  thing.Secretkey.set("!!!!!");
  key = thing.Secretkey.get();
  std::cout << "Secretkey = "
            << key
            << std::endl;

  thing.Secretkey = "?????";
  key = thing.Secretkey;
  std::cout << "Secretkey = "
            << key
            << std::endl;
  std::cout << std::endl;

  // and IndexedProperty<> members.
  // Multiple indices in square brackets
  // not supported yet
  thing.Assignments("Convenor",
                    "Herb");
  std::string job = thing.Assignments(
                                 "Convenor");
  std::cout << "Convenor = "
            << job
            << std::endl;

  thing.Assignments.set_Item("Minutes",
                             "Daveed");
  job = thing.Assignments.get_Item(
                                 "Minutes");
  std::cout << "Minutes = "
            << job
            << std::endl;

  thing.Assignments["Coffee"] = "Francis";
  job = thing.Assignments["Coffee"];
  std::cout << "Coffee = "
            << job
            << std::endl;
  std::cout << std::endl;

  return 0;
}
```