contents

credits & contacts

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

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

Advisors:

Phil Bass
phil@stoneymanor.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@gmail.com

Pippa Hennessy
pip@oldbat.co.uk

Advertising:
Thaddaeus Frogley
ads@accu.org

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

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

Information and Membership:
Join on the website or contact

David Hodge
membership@accu.org

Publications Officer:
John Merrells
publications@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial: Size Does Matter

The way that one goes about developing and delivering a software project depends critically on the scale of the project. There is no "one size fits all" approach. As a trivial example to illustrate this, no one would consider writing a test harness for a "hello world" program. (Actually, I have tried this question out on some TDD proponents over the last year - and I have only found one that insists that they would do so.)

Why shouldn't one write a test harness for "hello world"? As in all design questions it is a matter of trade-offs: there is a cost to doing it (a test harness for this program is typically more complex than the program itself) and a benefit (a test harness that confirms the program works). In this case the cost doesn't justify the benefit – as one respondent put it "I can just run the program to see that it works".

OK, you might say that is a silly example, but it reflects the basis on which the habits of those working in software development form. Their first programs *are* that simple. And they write them accordingly. As they become more accomplished they tackle bigger and bigger problems the same way – usually long past the point at which this is the most effective approach. Because they can still succeed the need for change isn't apparent to them. Typically it takes both a failure of catastrophic proportions, and also an open mind, before they appreciate the need for a different approach. Even so, old habits die hard and resisting the temptation to fix an urgent problem with "a quick 'low risk' hack" requires determination and vigilance.

This form of inertia (clinging to approaches that have become inappropriate) isn't restricted to the individual developer - one only has to look at the recent G8 response to climate change to see it operating at the scale of nations. But that example is outside the scope of this publication. What is relevant here is that it also applies to larger software development units: teams, departments and whole organisations.

The Scale of a Project

There are many ways to attempt to characterise the scale of a project:

- the number of developers;
- the value (and risk) to the business;
- the count of function points (or use cases, or stories…);
- the budget;
- the size of the codebase;
- the length of time the project is active;
- the range of technologies employed;
- Etc.

All of these affect the effectiveness of different methods of working. I hope no-one would claim that what is appropriate for one developer working on a low profile project for a few days is appropriate for a couple of dozen working on a high profile project for a couple of years. The choice of appropriate methods is an important decision and may need revision as a project progresses. Alastair Cockburn provides a lucid discussion of his research into the effect of scale on development process in "Agile Software Development".

It is very easy for an organisation that is accustomed to running projects of one size to continue to deploy the same practices on a project for which they are inappropriate. And, as with our developer in the first example, it often takes a significant failure before the assumptions that underlie this decision can be questioned. In fact, the practices are often habituated to an extent that means they are not even examined as a potential cause.

Opinions as to the cause of failure all too often fail to bear close examination, or are superficial – they identify a mistake by an individual without following up and discovering the circumstances that made that mistake inevitable. (There are many examples of this cited in "High-Pressure Steam Engines and Computer Software" - `http://www.safeware-eng.com/index.php/publications/HiPreStEn`.) If we look at those cases where there has been a thorough examination of failing projects it has been found that the individual errors of engineers and operators compounded inappropriate working and management practices. (Commonly cited examples include: Three Mile Island, Chernobyl, Challenger, Bhopal, and Flixborough.)

As Projects Grow

A typical development department will begin by delivering small projects, and its approach to the development and deployment of software is appropriate to this circumstance. In a small project everyone involved in development (which will often be the one and only developer) can be expected to have an understanding of each of the elements of the project, their interactions and the context in which these elements work. They will also have an understanding of the status of any ongoing work.

Sooner or later the department will begin to undertake medium (or even large) scale projects - I'll explain what I mean by "medium" and "large" in a moment. The point I want to make first is that there is nothing obvious to alert an organisation that a new project with an extra developer or two, lasting a little longer, using an extra technology or two, and with a bit more functionality than usual requires a different approach.

It would be great to have a rule that took the size of a project and gave the appropriate development practices. Sadly, it isn't that simple. There are just too many factors affecting both the size of the project and the level of risk that an organisation is prepared to

accept. In practice, I find that it is only by observing as the project progresses that a useful measure emerges. But it is rare to find an organisation that will take early indicators on board and make changes *when it is not clear that anything will go wrong*.

Small, Medium or Large

The distinction I use is based upon whether the whole project will "fit in the head" of all developers (small project), a few of the developers (medium project), or none of the developers (large project). It may not be immediately apparent, but a project can move between these categories during its lifecycle - and not just in one direction. (One project I am aware of moved from small to medium to large and then back to medium, and if the current initiatives succeed may yet make it back to small.)

In a typical medium sized project there will be one or more people with this general understanding and a number of people with specialised knowledge of particular areas. It is possible to run such a project without taking account of these specialisations without a big risk of disaster. Disaster can happen when firstly, a subsystem reaches a state that requires specialised knowledge to work on it; secondly, that this specialised knowledge is lost to the team; and thirdly, work is undertaken that involves that subsystem. There are development practices that mitigate these risks but, like any insurance scheme, these have a cost that eliminates them from consideration on a small project.

In a large project no-one will have an understanding of the whole thing in detail – those with a general understanding of the overall structure will rely on the expertise of those with a detailed knowledge of individual components and vice versa. Any significant development activity will cross boundaries of expertise - and, in consequence, will entail a risk of changing things without the specialised knowledge that makes the change safe. (There are ways to mitigate this risk, but not with a "small project" mentality.)

It is rare, therefore, to find a "large project" running smoothly with "small project" practices - typically many pieces of work will be failing. But even with this clue that something is systemically wrong the reaction is often inappropriate - after all the developer(s) implementing any piece of work that went wrong clearly made mistakes. And this brings me back to the arguments presented in "High-Pressure Steam Engines and Computer Software" (this really is a good paper - if you are not familiar with it, go and read it). Working practices should be chosen to both minimise the likelihood of mistakes and to ensure that any mistakes are detected and corrected before they derail the project.

Changing Working Practices

So what do you do when you are on a project that is showing symptoms of a mismatch between the working practices and the nature of the project?

If you are a single developer working on a three-day project then it is probably easy to decide not to allocate work based on "SecondBestResource". (Indeed, if you succeed in employing this pattern, then you probably have worse problems than the project failing!) But problems can be subtle - is the cost of setting up and maintaining a build server for the project really justified? (Even if it is required for conformance to departmental policy!)

On a larger project it is much harder to institute change - not least because changes need to be negotiated with other project members (who will not be inclined to change unless you first convince them that there is a need). But even when you've successfully convinced everyone that a build server would be a good idea *someone needs to spend time setting it up and maintaining it*. And this is often the sticking point - there are "brownie points" to be had implementing features the customer will use, and the customer doesn't care less about version control, test infrastructure, or the internal review of work items. In these circumstances who would want to be seen spending time on this stuff? It requires co-operation to tackle these issues.

Strategies for Co-operation

There are two basic strategies for co-operation: either someone takes responsibility and orchestrates the activities of others, or everyone takes responsibility for dealing with the problems she or he identifies.

Both can work for small and medium sized projects and, in many cases, it is easier to get one person to take responsibility than to ensure that everyone does - which can make the first strategy easier to implement. However, as the size of a project increases, it becomes harder and harder for one person to keep track of everything that needs doing and on large projects it becomes impossible. There are, of course, ways to scale the first strategy - break down the project's issues into groups (by sub-team, by technology, by geography, or whatever) and ensure that someone takes responsibility for each group. However, this always seems to leave some issues that everyone disowns.

The strategy of everyone taking responsibility does scale a lot better if everyone co-operates. The difficulty is getting everyone to "buy into" this approach to begin with. It takes trust - and at the beginning of a project this has typically not been earned. It can be very difficult to convince everyone that "freeloaders" will not be a problem - until they've participated in a team that works this way. The thing that is missed is that the team is a small enough social unit that "freeloaders" are quickly identified and dealt with along with other problems.

A Personal Strategy

As a member of a project one should behave as one believes others ought to behave. The worst thing that can be done on encountering a problem is to ignore it on the basis that "someone else" should deal with it. The next worst thing is to bury it in a write-only "issues list" in the hope that one day someone will deal with it. If everyone behaves like that then nobody deals with anything.

Everyone - including you and me - who encounters a problem has a responsibility to do something with it: either deal with it, or find someone better qualified to agrees to take responsibility.

Alan Griffiths
overload@accu.org

Copy Deadlines

All articles intended for publication in *Overload 69* should be submitted to the editor by September 1st 2005, and for *Overload 70* by November 1st 2005.

C++ Interface Classes – Noise Reduction

by Mark Radford

Interface classes are a principle mechanism for separating a class' interface from its implementation in C++. I wrote an introduction to interface classes in a previous article [1], and Alan Griffiths and I included the technique in our survey of techniques for separating interface and implementation in C++ [2].

In this article, I intend to explore interface classes – and their *implementation classes* – further. The topics I plan to cover are:

- How interface and implementation classes can be designed into the code in such a way as to reduce implementation "noise"
- How factory functions can be used to facilitate the above
- A way of managing instance lifecycles when factory functions are used to encapsulate different memory allocation mechanisms

An Example Class

In [2] Alan and I used `telephone_list` – a telephone address book class – in order to illustrate several C++ interface/implementation separation techniques. Here I will again use (a slightly modified version of) that example.

The `telephone_list` interface class looks like this:

```
class telephone_list
{
public:
  virtual ~telephone_list()   {}

  virtual std::string name() const = 0;

  virtual std::pair<bool, std::string>
  number(const std::string& person) const = 0;

  virtual telephone_list&
  add_entry(const std::string& name,
            const std::string& number) = 0;

protected:
  telephone_list() {}
  telephone_list(const telephone_list& rhs) {}
private:
  telephone_list& operator=(const
telephone_list& rhs);
};
```

In order for this to have functionality, and in order for instances to be created, an implementation class is needed – I'm going to call it `telephone_list_imp`:

```
class telephone_list_imp : public
telephone_list
{
public:
  telephone_list_imp(const std::string&
list_name);

private:

  virtual ~telephone_list_imp();

  virtual std::string name() const;

  virtual std::pair<bool, std::string>
  number(const std::string& person) const;

  virtual telephone_list&
  add_entry(const std::string& name,
            const std::string& number);

  typedef std::map<std::string, std::string>
dictionary_t;

  std::string  name_rep;
  dictionary_t dictionary_rep;

  telephone_list_imp(const telephone_list_imp&
rhs);
  telephone_list_imp& operator=(const
telephone_list_imp& rhs);
};
```

In [1] I also described implementation only classes, and this is an approach I have applied here. Apart from the constructors, all member functions have been made private. This strengthens the separation of interface from implementation by making it possible to create instances of `telephone_list_imp`, while usage must be via pointers and/or references to `telephone_list`.

Hiding the Implementation and Creating Instances

This whole design is geared up to functionality being used through pointers/references to `telephone_list`. Therefore, the only reason to make the definition of `telephone_list_imp` visible to client code is so that instances can be created. It follows that client code has to carry a certain amount of "noise" – in the form of the publicly visible definition of `telephone_list_imp` – just so instances can be created.

Further, C++ has the problem of physical dependencies between header files, and the consequent recompilations that result from modifications being made to them. This is a consequence of the file inclusion model inherited from C. Let's say for the sake of an example, that one day `telephone_list_imp` is modified, abandoning the `std::map` implementation in favour of a different container. The fact that client code – which has no dependency on the modified implementation detail – needs to recompile, emphasises the fact that `telephone_list_imp` is just noise to the client code.

The two issues discussed above add up to the fact that it would be better if `telephone_list_imp`'s definition could be kept out of client code altogether. Ideally, the best place for the definition of `telephone_list_imp` is in an implementation (typically `.cpp`) file. This leads to another problem of how clients can create instances, but this is straightforward to solve: in the header file, provide a factory function for creating instances of `telephone_list_imp`. The header file `telephone_list_imp.h` (with include "guards" removed for brevity) now looks like this:

```
#include "telephone_list.h"
#include <string>
telephone_list* create_telephone_list(const
    std::string& list_name);
```

Note that there is no mention of the implementation class – the `telephone_list` interface class is all that's needed. In fact, only a forward declaration of `telephone_list` is needed – however, the header file has been included because users might reasonably expect that when they write `#include "telephone_list_imp.h"` in client code, the base class' definition will be made available.

The fragment of the implementation file containing the `telephone_list_imp` and factory function definition looks like this:

```
class telephone_list_imp : public
telephone_list
{
public:
  telephone_list_imp(const std::string&
       list_name);

private:

  virtual ~telephone_list_imp();

  virtual std::string name() const;

  virtual std::pair<bool, std::string>
      number(const std::string& person) const;

  virtual telephone_list&
  add_entry(const std::string& name,
            const std::string& number);

  typedef std::map<std::string, std::string>
        container;

  std::string  name_rep;
  container dictionary_rep;

  telephone_list_imp
        (const telephone_list_imp& rhs);
  telephone_list_imp& operator=
        (const telephone_list_imp& rhs);
};

telephone_list* create_telephone_list
        (const std::string& list_name)
{
  return new telephone_list_imp(list_name);
}
...
```

At this point in the exercise the aim of removing `telephone_list`'s implementation from having visibility in client code has been achieved. Clients deal with pointers/references to `telephone_lists`, while `telephone_list_imp` remains buried safely in its implementation file. All should be well, but the solution to one problem has created another.

How are Instances Deleted?

There are two observations to make about the ("naïve") implementation of `create_telephone_list()`:

- The mechanism used to create instances is now encapsulated and hidden from public view
- So far, the return type is a simple pointer to telephone_list

This means clients can apply the delete operator to pointers returned from `create_telephone_list()`. However, they have to rely on documentation to know they must do this. There is no way it can be made clear in the code, and clients can't assume it because using the delete operator is not compatible with all mechanisms for allocating class instances on the heap in C++. A solution to the problem (not the only one) is, rather than return a simple pointer, to return a smart pointer such as Boost's `shared_ptr` (see [3]). The (fragmented form of the) header file `telephone_list_imp.h` now looks like this:

```
#include "telephone_list.h"
#include "boost/shared_ptr.hpp"
#include <string>

boost::shared_ptr<telephone_list>
create_telephone_list(const std::string&
list_name);
```

While the implementation of `create_telephone_list()` now looks like this:

```
boost::shared_ptr<telephone_list>
create_telephone_list(const std::string&
list_name)
{
  telephone_list* p = new
telephone_list_imp(list_name);
  return boost::shared_ptr<telephone_list>(p);
}
```

In passing note the avoidance of the expression:

```
return boost::shared_ptr<telephone_list>(new
telephone_list_imp(list_name));
```

This is because `boost::shared_ptr` "remembers" the concrete type created, and uses it when the instance is deleted – i.e. `telephone_list_imp` having a private destructor means `boost::shared_ptr`'s attempt to delete via it causes a compile error. The mechanism used ensures that the "remembered" type is `telephone_list`, and thus avoids compilation problems. Another option is simply to make `telephone_list_imp`'s destructor public. I chose the option in the code fragment because it adheres to the principle of all usage being through the interface class.

The above approach solves the problem of deleting the instance that had its creation mechanism encapsulated. The cost of achieving this is the hard-wiring of a specific smart pointer into the code. Further, there is a remaining problem that it doesn't solve.

Different Allocation Mechanisms

The memory allocation scheme used so far is not the only one available in C++. For example, the placement form of `new` could be used to construct instances in conjunction with using `malloc()` to allocate the memory. However, if `create_telephone_list()` returns a simple pointer and relies on the client code to apply the `delete` operator, then there's no way its implementation can ever be changed to use an alternative allocation mechanism.

In some design scenarios, as well as having a factory function to create instances, it is possible to have a disposal function to delete them. However in the design scenario under consideration, there is a serious drawback to this approach. The implementation class `telephone_list_imp` is implemented in a way that results in

particular complexity characteristics – i.e. those associated with its implementation container `std::map`. Imagine that there arises a need for a second implementation with different characteristics. Why this may be so is outside the scope of this article – suffice to say that if this is done, `telephone_list_imp` ceases to be the only implementation of `telephone_list` in town. Getting back to disposing of instances, it is not hard to see that in order for clients to pass instances to a disposal function, either instances of each implementation class would need to use the same memory allocation mechanism, or disposal functions would need some way of recovering the implementation class from a pointer/reference to the interface class.

The analysis of the complexities and tradeoffs involved in using disposal function may at some point be the subject of another article, but in this one I want to look at a different approach. The approach I want to look at involves associating a disposal function with class instances at the time their factory function creates them. Here, just for illustration's sake, is a fragment of a home grown smart pointer that achieves this:

```
template <typename T> class ref_counted_ptr
{
public:
  ref_counted_ptr(T* p, void (*delete_fn)(T*))
    : pointee(p),
      del(delete_fn)
  {
     ...
  }

  ~ref_counted_ptr() { del(pointee); }

  T* operator->()
  {
    return pointee;
  }
...
private:
  T* pointee;
  void (*del)(T*);
...
};
```

Using `ref_counted_ptr`, the declaration of the factory function now looks like this:

```
ref_counted_ptr<telephone_list>
create_telephone_list(const std::string&
list_name);
```

Its implementation now looks like this:

```
ref_counted_ptr<telephone_list>
      create_telephone_list(const std::string&
      list_name)
{
  telephone_list* mem =
      static_cast<telephone_list*>(std::malloc
      (sizeof telephone_list_imp));

  if (!mem)
    throw std::bad_alloc();

  telephone_list* pobj = new (mem)
telephone_list_imp(list_name);

  return ref_counted_ptr<telephone_list>(pobj,
del_telephone_list);
}
```

As if by magic, a function called `del_telephone_list()` has appeared in the above code fragment – it looks like this:

```
void del_telephone_list(telephone_list* p)
{
  p->~telephone_list();
  std::free(p);
}
```

However, as I said, `ref_counted_ptr` is for illustration only. There is actually no need to write a custom smart pointer just to associate a disposal function with a class instance, because `boost::shared_ptr` has a mechanism for accommodating a disposal function. Actually, `boost::shared_ptr` has a somewhat more sophisticated mechanism that allows the disposal function to be either a pointer to a function, or a function object. For this article, I'll stick to the approach already used – that of using `del_telephone_list()` as shown above. The factory function implementation now looks like this:

```
boost::shared_ptr<telephone_list>
create_telephone_list(const std::string&
list_name)
{
  telephone_list* mem =

static_cast<telephone_list*>(std::malloc(sizeo
f telephone_list_imp));

  if (!mem)
    throw std::bad_alloc();

  telephone_list* pobj = new (mem)
telephone_list_imp(list_name);

  return
boost::shared_ptr<telephone_list>(pobj,
del_telephone_list);
}
```

In passing I should mention that there are various tradeoffs in possible implementations of reference counted smart pointers, and `boost::shared_ptr` addresses only one set of tradeoffs. I just thought it best to point that out; sorry, but I'm not going into any more detail on that topic. The reader is referred to the Boost documentation (see [3]).

Finally

Cases where solutions to problems are the solutions are rare – usually there are alternatives that come with their own sets of tradeoffs. I hope I have succeeded in making the tradeoffs clear. This article has covered the three points set out in the

[concluded at foot of next page]

A Technique for Register Access in C++

Pete Goodliffe

Exploiting C++'s features for efficient and safe hardware register access.

This article originally appeared in C/C++ Users Journal, and is reproduced by kind permission.

Embedded programmers traditionally use C as their language of choice. And why not? It's lean and efficient, and allows you to get as close to the metal as you want. Of course C++, used properly, provides the same level of efficiency as the best C code. But we can also leverage powerful C++ features to write cleaner, safer, more elegant low-level code. This article demonstrates this by discussing a C++ scheme for accessing hardware registers in an optimal way.

Demystifying Register Access

Embedded programming is often seen as black magic by those not initiated into the cult. It does require a slightly different mindset; a resource constrained environment needs small, lean code to get the most out of a slow processor or a tight memory limit. To understand the approach I present we'll first review the mechanisms for register access in such an environment. Hardcore embedded developers can probably skip ahead; otherwise here's the view from 10,000 feet.

Most embedded code needs to service hardware directly. This seemingly magical act is not that hard at all. Some kinds of register need a little more fiddling to get at than others, but you certainly don't need an eye-of-newt or any voodoo dances. The exact mechanism depends on how your circuit board is wired up. The common types of register access are:

- **Memory mapped I/O** The hardware allows us to communicate with a device using the same instructions as memory access. The device is wired up to live at memory address *n*; register 1 is mapped at address *n*, register 2 is at *n*+1, register 3 at *n*+2, and so on.
- **Port mapped I/O** Certain devices present pages of registers that you have to map into memory by selecting the correct device 'port'. You might use specific input/output CPU instructions to talk to these devices, although more often the port and its selector are mapped directly into the memory address space.
- **Bus separated** It's harder to control devices connected over a non-memory mapped bus. I^2C and I^2S are common peripheral connection buses. In this scenario you must either talk to a dedicated I^2C control chip (whose registers *are* memory mapped), telling it what to send to the device, or you manipulate I^2C control lines yourself using GPIO[1] ports on some other memory mapped device.

Offset	Size	Name	R/W	Description
+0x00	1 byte	STATUS	RW	**UART status register**. Bits: 0: TX buffer has empty space 1: RX buffer has empty space 2: Transmit underrun 3: Receive overflow Write 1 to bits 2 or 3 to clear a status report.
+0x01	1 byte	TXCTL	RW	**Transmit control**. Bits: 0: enable transmitter 1-3: no of bytes in transmit buffer to send
+0x02	1 byte	RXCTL	RW	**Receive control**. Bits: 0: enable receiver 1-3: no of bytes in receive buffer to be read
+0x04	4 bytes	TXBUF	W	**Transmit buffer**. 0-7: Byte 1 8-15: Byte 2 16-23: Byte 3 24-31: Byte 4
+0x08	4 bytes	RXBUF	R	**Receive buffer**. 0-7: Byte 1 8-15: Byte 2 16-23: Byte 3 24-31: Byte 4 Reading this register clears the buffer, and resets the RXCTL count.

Device base address is 0xfffe0000.
Register offsets are byte indexes from this base.

Figure 1: Registers in a sample UART line driver device

[continued from previous page]

introduction, having followed one train of thought. Others have been alluded to in passing but not covered, but perhaps in future articles.

Mark Radford
mark@twonine.co.uk

References

1 Mark Radford, *C++ Interface Classes – An Introduction* (Overload 62, and also available from http://www.twonine.co.uk/articles/CPPInterfaceClassesIntro.pdf)

2 Alan Griffiths and Mark Radford, *Separating Interface and Implementation in C++*, (Overload, and also available at http://www.twonine.co.uk/articles/SeparatingInterfaceAndImplementation.pdf)

3 www.boost.org

1 General Purpose Input/Output - assignable control lines not specifically designed for a particular data bus.

Each device has a data sheet that describes (amongst other things) the registers it contains, what they do, and how to use them. Registers are a fixed number of bits wide - this is usually determined by the type of device you are using. This is an important fact to know: some devices will lock up if you write the wrong width data to them. With fixed width registers, many devices cram several bits of functionality into one register as a 'bitset'. The data sheet would describe this diagrammatically in a similar manner to Figure 1.

So what does hardware access code look like? Using the simple example of a fictional UART line driver device presented in Figure 1, the traditional C-style schemes are:

- **Direct memory pointer access**. It's not unheard of to see register access code like Listing 1, but we all know that the perpetrators of this kind of monstrosity should be taken outside and slowly tortured. It's neither readable nor maintainable.

  Pointer usage is usually made bearable by defining a macro name for each register location. There are two distinct macro flavours. The first macro style defines bare memory addresses (as in Listing 2). The only real advantage of this is that you can share the definition with assembly code parsed using the C preprocessor. As you can see, its use is long-winded in normal C code, and prone to error - you have to get the cast right each time. The alternative, in Listing 3, is to include the cast in the macro itself; far nicer in C. Unless there's a lot of assembly code this latter approach is preferable.

  We use macros because they have no overhead in terms of code speed or size. The alternative, creating a physical pointer variable to describe each register location, would have a negative impact on both code performance and executable size. However, macros are gross and C++ programmers are already smelling a rat here. There are plenty of problems with this fragile scheme. It's programming at a very low level, and the code's real intent is not clear- it's hard to spot all register accesses as you browse a function.
- **Deferred assignment** is a cute technique that allows you to write code like Listing 4, defining the register location values at link time. This is not commonly used; it's cumbersome when you have a number of large devices, and not all compilers provide this functionality. It requires you to run a flat (non virtual) memory model.
- Use a `struct` to describe the register layout in memory, as in Listing 5. There's a lot to be said for this approach - it's logical and reasonably readable. However, it has one big drawback: it is not standards-compliant. Neither the C nor C++ standards specify how the contents of a struct are laid out in memory. You

```
*((volatile uint32_t *)0xfffe0004) = 10;
*((volatile uint8_t  *)0xfffe0001) = 3;
```

Listing 1

```
#define UART_TXBUF 0xfffe0004
#define UART_TXCTL 0xfffe0001
*(volatile uint32_t *)UART_TXBUF = 10;
*(volatile uint8_t  *)UART_TXCTL = 3;
```

Listing 2

```
#define UART_TXBUF ((volatile uint32_t*) 0xfffe0004)
#define UART_TXCTL ((volatile uint8_t*)  0xfffe0001)
*UART_TXBUF = 10;
*UART_TXCTL = 3;
```

Listing 3

```
extern volatile uint32_t UART_TXBUF;
extern volatile uint8_t  UART_TXCTL;
UART_TXBUF = 10;
UART_TXCTL = 3;

// compile this with:
//   gcc listing4.c
//     -gUART_UART_TXBUF=0xfffe0004
//     -gUART_TXCTL=0xfffe0001
```

Listing 4

```
struct uart_device_t
{
  uint8_t STATUS;
  uint8_t TXCTL;
 .. and so on ...
};
static volatile uart_device_t *
    const uart_device
    = reinterpret_cast
    <volatile uart_device_t *>(0xfffe0000);
uart_device->TXBUF = 10;
uart_device->TXCTL = 3;
```

Listing 5

```
#define UART_RX_BYTES 0x0e
uint32_t uart_read()
{
  while ((*UART_RXCTL & UART_RX_BYTES) == 0)
      // manipulate here
  {
    ; // wait
  }
  return *UART_RXBUF;
}
```

Listing 6

Using Volatile

This low-level purgatory is where we use C's volatile keyword. volatile signals to the compiler that a value may change under the code's feet, that we can make no assumptions about it, and the optimiser can't cache it for repeated use.

This is just the behaviour we need for hardware register access. Every time we write code that accesses a register we want it to result in a real register access. Don't forget the volatile qualification!

are guaranteed an exact ordering, but you don't know how the compiler will pad out non-aligned items. Indeed, some compilers have proprietary extensions or switches to determine this behaviour. Your code might work fine with one compiler and produce startling results on another.

- Create a **function** to access the registers and hide all the gross stuff in there. On less speedy devices this might be prohibitively slow, but for most applications it is perfectly adequate, especially for registers that are accessed infrequently. For port mapped registers this makes a lot of sense; their access requires complex logic, and writing all this out longhand is tortuous and easy to get wrong.

It remains for us see how to manipulate registers containing a bitset. Conventionally we write such code by hand, something like Listing 6. This is a sure-fire way to cause yourself untold grief, tracking down odd device behaviour. It's very easy to manipulate the wrong bit and get very confusing results.

Does all this sound messy and error prone? Welcome to the world of hardware devices. And this is just addressing the device: what you write into the registers is your own business, and part of what makes device control so painful. Data sheets are often ambiguous or miss essential information, and devices magically require registers to be accessed in a certain order. There will never be a silver bullet and you'll always have to wrestle these demons. All I can promise is to make the fight less biased to the hardware's side.

A More Modern Solution

So having seen the state of the art, at least in the C world, how can we move into the 21st century? Being good C++ citizens we'd ideally avoid all that nasty preprocessor use and find a way to insulate us from our own stupidity. By the end of the article you'll have seen how to do all this and more. The real beauty of the following scheme is its simplicity. It's a solid, proven approach and has been used for the last five years in production code deployed in tens of thousands of units across three continents.

Here's the recipe…

Step one is to junk the whole preprocessor macro scheme, and define the device's registers in a good old-fashioned enumeration. For the moment we'll call this enumeration Register. We immediately lose the ability to share definitions with assembly code, but this was never a compelling benefit anyway. The enumeration values are specified as offsets from the device's base memory address. This is how they are presented in the device's datasheet, which makes it easier to check for validity. Some data sheets show byte offsets from the base address (so 32-bit register offsets increment by 4 each time), whilst others show 'word' offsets (so 32-bit register offsets increment by 1 each time). For simplicity, we'll write the enumeration values however the datasheet works.

The next step is to write an inline regAddress function that converts the enumeration to a physical address. This function will be a very simple calculation determined by the type of offset in the enumeration. For the moment we'll presume that the device is memory mapped at a known fixed address. This implies the simplest MMU configuration, with no virtual memory address space in operation. This mode of operation is not at all uncommon in embedded devices. Putting all this together results in Listing 7.

```
static const unsigned int baseAddress =
      0xfffe0000;
enum Registers
{
  STATUS = 0x00, // UART status register
  TXCTL  = 0x01, // Transmit control
  RXCTL  = 0x02, // Receive control
  . and so on ...
};
inline volatile uint8_t *regAddress
      (Registers reg)
{
  return reinterpret_cast<volatile
        uint8_t*>(baseAddress + reg);
}
```

Listing 7

The missing part of this jigsaw puzzle is the method of reading/writing registers. We'll do this with two simple inline functions, `regRead` and `regWrite`, shown in Listing 8. Being inline, all these functions can work together to make neat, readable register access code with no runtime overhead whatsoever. That's mildly impressive, but we can do so much more.

Different Width Registers

Up until this point you could achieve the same effect in C with judicious use of macros. We've not yet written anything groundbreaking. But if our device has some 8-bit registers and some 32-bit registers we can describe each set in a different enumeration. Let's imaginatively call these Register8 and Register32. Thanks to C++'s strong typing of enums, now we can overload the register access functions, as demonstrated in Listing 9.

Now things are getting interesting: we still need only type `regRead` to access a register, but the compiler will automatically ensure that we get the correct width register access. The only way to do this in C is manually, by defining multiple read/write macros and selecting the correct one by hand each time. This overloading shifts the onus of knowing which registers require 8 or 32-bit writes from the programmer using the device to the compiler. A whole class of error silently disappears. Marvellous!

Extending to Multiple Devices

An embedded system is composed of many separate devices, each performing their allotted task. Perhaps you have a UART for control, a network chip for communication, a sound device

```
inline uint8_t regRead(Registers reg)
{
  return *regAddress(reg);
}

inline void regWrite
      (Registers reg, uint8_t value)
{
  *regAddress(reg) = value;
}
```

Listing 8

for audible warnings, and more. We need to define multiple register sets with different base addresses and associated bitset definitions. Some large devices (like super I/O chips) consist of several subsystems that work independently of one another; we'd also like to keep the register definitions for these parts distinct.

The classic C technique is to augment each block of register definition names with a logical prefix. For example, we'd define the UART transmit buffer like this:

```
#define MYDEVICE_UART_TXBUF
      ((volatile uint32_t *)0xffe0004)
```

C++ provides an ideal replacement mechanism that solves more than just this aesthetic blight. We can group register definitions within namespaces. The nest of underscored names is replaced by `::` qualifications - a better, syntactic indication of relationship. Because the overload rules honour namespaces, we can never write a register value to the wrong device block: it's a syntactic error. This is a simple trick, but it makes the scheme incredibly usable and powerful.

```
// New enums for each register width
enum Registers8
{
  STATUS = 0x00, // UART status register
  ... and so on ...
};
enum Registers32
{
  TXBUF = 0x04,  // Transmit buffer
    ... and so on ...
};

// Two overloads of regAddress
inline volatile uint8_t *regAddress
      (Registers8 reg)
{
  return reinterpret_cast<volatile uint8_t*>
        (baseAddress + reg);
}
inline volatile uint32_t *regAddress
      (Registers32 reg)
{
  return reinterpret_cast<volatile uint32_t*>
        (baseAddress + reg);
}
// Two overloads of regRead

inline uint8_t regRead(Registers8 reg)
{
  return *regAddress(reg);
}
inline uint32_t regRead(Registers32 reg)
{
  return *regAddress(reg);
}
..similarly for regWrite ...
```

Listing 9

Proof of Efficiency

Perhaps you think that this is an obviously a good solution, or you're just presuming that I'm right. However, a lot of old-school embedded programmers are not so easily persuaded. When I introduced this scheme in one company I met a lot of resistance from C programmers who could just not believe that the inline functions resulted in code as efficient as the proven macro technique.

The only way to persuade them was with hard data - I compiled equivalent code using both techniques for the target platform (gcc targeting a MIPS device). The results are listed in the table below. An inspection of the machine code generated for each kind of register access showed that the code was *identical*. You can't argue with that!

Register access method	Results (object file size in bytes)	
	Unoptimised	Optimised
C++ inline function scheme	1087	551
C++ using #defines	604	551
C using #defines	612	588

It's particularly interesting to note that the #define method in C is slightly larger than the C++ equivalent. This is a peculiarity of the gcc toolchain - the assembly listing for the two main functions is identical: the difference in file size is down to the glue around the function code.

Namespacing also allows us to write more readable code with a judicious sprinkling of using declarations inside device setup functions. Koenig lookup combats excess verbiage in our code. If we have register sets in two namespaces `DevA` and `DevB`, we needn't quality a `regRead` call, just the register name. The compiler can infer the correct `regRead` overload in the correct namespace from its parameter type. You only have to write:

```
uint32_t value = regRead(DevA::MYREGISTER);
    // note: not DevA::regRead(...)
```

Variable Base Addresses

Not every operating environment is as simplistic as we've seen so far. If a virtual memory system is in use then you can't directly access the physical memory mapped locations - they are hidden behind the virtual address space. Fortunately, every OS provides a mechanism to map known physical memory locations into the current process' virtual address space.

A simple modification allows us to accommodate this memory indirection. We must change the `baseAddress` variable from a simple static const pointer to a real variable. The header file defines it as extern, and before any register accesses you must arrange to define and assign it in your code. The definition of `baseAddress` will be necessarily system specific.

Other Usage

Here are a few extra considerations for the use of this register access scheme:

- Just as we use namespaces to separate device definitions, it's a good idea to choose header file names that reflect the logical

device relationships. It's best to nest the headers in directories corresponding to the namespace names.

- A real bonus of this register access scheme is that you can easily substitute alternative `regRead/regWrite` implementations. It's easy to extend your code to add register access logging, for example. I have used this technique to successfully debug hardware problems. Alternatively, you can set a breakpoint on register access, or introduce a brief delay after each write (this quick change shows whether a device needs a pause to action each register assignment).
- It's important to understand that this scheme leads to larger *unoptimised* builds. Although it's remarkably rare to not optimise your code, without optimisation inline functions are not reduced and your code will grow.
- There are still ways to abuse this scheme. You can pass the wrong bitset to the wrong register, for example. But it's an order of magnitude harder to get anything wrong.
- A small sprinkling of template code allows us to avoid repeated definition of `bitRead/bitWrite`. This is shown in Listing 11.

Conclusion

OK, this isn't rocket science, and there's no scary template metaprogramming in sight (which, if you've seen the average embedded programmer, is no bad thing!) But this is a robust technique that exploits a number of C++'s features to provide safe and efficient hardware register access. Not only is it supremely readable and natural in the C++ idiom, it prevents many common register access bugs and provides extreme flexibility for hardware access tracing and debugging.

I have a number of proto-extensions to this scheme to make it more generic (using a healthy dose of template metaprogramming, amongst other things). I'll gladly share these ideas on request, but would welcome some discussion about this.

Do Overload readers see any ways that this scheme could be extended to make it simpler and easier to use?

Pete Goodliffe
<pete@cthree.org>

Pete Goodliffe is a senior C++ programmer who never stays at the same place in the software food chain. An ACCU columnist, he has a passion for curry and doesn't wear shoes.

```
// A macro that defines enumeration values for
// a bitset
// You supply the start and end bit positions

#define REG_BIT_DEFN(start, end)
       ((start<<16)|(end-start+1))

enum STATUS_bits
{
  TX_BUFFER_EMPTY = REG_BIT_DEFN(0, 0),
  RX_BUFFER_EMPTY = REG_BIT_DEFN(1, 1),
  TX_UNDERRUN     = REG_BIT_DEFN(2, 2),
  RX_OVERFLOW     = REG_BIT_DEFN(3, 3)
};
... similarly for other bitsets ...

#undef REG_BIT_DEFN

inline uint32_t bitRead(Registers32 reg,
                        uint32_t bits)
{
  uint32_t       regval = *regAddress(reg);
  const uint32_t width  = bits & 0xff;
  const uint32_t bitno  = bits >> 16;
  regval >>= bitno;
  regval  &= ((1<<width)-1);
  return regval;
}

inline void bitWrite(Registers32 reg,
                     uint32_t bits,
                     uint32_t value)
{
  uint32_t          regval = *regAddress(reg);
  const uint32_t    width  = bits & 0xff;
  const uint32_t    bitno  = bits >> 16;
  regval &= ~(((1<<width)-1) << bitno);
  regval |=  value << bitno;
  *regAddress(reg) = regval;
}
```

Listing 10

```
// Template versions of bitRead/Write - put
// them at global scope and you don't have to
// copy bitRead/Write into every device
// namespace

template <typename RegType>
inline uint32_t bitRead
      (RegType reg, uint32_t bits)
{
  uint32_t       regval = *regAddress(reg);
  const uint32_t width  = bits & 0xff;
  const uint32_t bitno  = bits >> 16;
  regval >>= bitno;
  regval  &= ((1<<width)-1);
  return regval;
}
template <typename RegType>
inline void bitWrite(RegType reg,
      uint32_t bits, uint32_t value)
{
  uint32_t          regval = *regAddress(reg);
  const uint32_t    width  = bits & 0xff;
  const uint32_t    bitno  = bits >> 16;
  regval &= ~(((1<<width)-1) << bitno);
  regval |=  value << bitno;
  *regAddress(reg) = regval;
}
```

Listing 11

Investigating Java Class Loading

by Roger Orr

Introduction

The class loader in Java is a powerful concept that helps to provide an extensible environment for running Java code with varying degrees of trust. Each piece of byte code in a running program is loaded into the Java virtual machine by a class loader, and Java can grant different security permissions to runtime objects based upon the class loaders used to load them.

Most of the time this mechanism is used implicitly by both the writer and user of a Java program and 'it just works'. However there is quite a lot happening behind the scenes; for example when you run a Java applet some of the classes are being loaded across the Internet while others are read from the local machine. The class loader does the work of getting the byte code from the target Web site and it also helps to enforce the so-called 'sandbox' security model.

> Java provides a security model known as the 'sandbox model' where untrusted code executes in its own environment with no risk of doing any damage to the full environment. All attempts to 'get out of the sandbox', such as opening files on the local machine or issuing network requests to arbitrary ports, are first checked by a security manager assigned to the JVM (Java Virtual Machine). The security manager can thereby provide complete control over the level of access the running program has to the rest of the machine. This makes it relatively safe to run code, such as a Java applet, which may be downloaded from a remote Web site over which you have no control as the security manager can validate the downloaded code and restrict its access to predefined sets of actions.

Another place where class loaders are used is for Web services. Typically the main application classes are loaded from a WAR (Web ARchive) file but may make use of standard Java classes as well as other classes or JAR (Java ARchive) files that may be shared between multiple applications running inside a single server. In this case the principal reason for the extra class loaders is to ensure that each Web application remains as independent of the others as possible and in particular that there is no conflict should a class with the same name exist in two different WAR files. Java achieves this because each class loader defines a separate name space - two Java classes are the same only if they were loaded with the same class loader. As we shall see this can have some surprising results.

Using an Additional Class Loader

In the case of a browser or a Web server the framework usually provides all the various class loaders. However you can use additional class loaders, and it is surprisingly easy to do so. Java provides an abstract base class, java.lang.ClasssLoader, which all class loaders must extend. The normal model is that each class loader has a link to its 'parent' class loader and all requests for loading classes are first passed to the parent to see if they can be loaded, and only if this delegated load fails does the class loader try to satisfy the load. The class loaders for a Java program form a tree, with the 'bootstrap' class loader as the top node of the tree and this model ensures that standard Java classes, such as String, are found in the usual place and only the application's own classes are loaded with the user-supplied class handler. (Note that this is only a convention and not all class loaders follow the same pattern. In particular it is up to the implementer of a class loader to decide when and if to delegate load requests to its parent)

One important issue when creating a class loader is deciding which class loader to use as the parent. There are several possibilities:

- No parent loader. In this case the loader will be responsible for loading *all* classes.
- Use the system class loader. This is the commonest practice.
- Use the class loader used to load the current class. This is how Java itself loads dependent classes.
- Use a class loader for the current thread context.

Java provides a simple API for getting and setting the default class loader for the current thread context. This can be useful since Java does not provide any way to navigate from a parent class loader to its child class loader(s). I demonstrate setting the thread's default class loader in the example below.

Java provides a standard URLClassLoader that is ready to use, or you can implement your own class loader.

As an example of the first case, you might want to run a Java program on workstations in your organisation, but be able to hold all the Java code centrally on a Web server. Here is some example code that uses the standard java.net.URLClassLoader to instantiate an object from a class held, in this instance, on my own Web site:

```
/**
 *This is a trivial example of a class loader.
 *It loads an object from a class on my own
 *Web site.
 */

public class URLExample
{
  private static final String defaultURL =
        "http://www.howzatt.demon.co.uk/";
  private static final String defaultClass =
       "articles.java.Welcome";

  public static void main( String args[] )
       throws Exception
  {
    final String targetURL = ( args.length
          < 1 ) ? defaultURL : args[0];
    final String targetClass = ( args.length
          < 2 ) ? defaultClass : args[1];

    // Step 1: create the URL class loader.
    System.out.println( "Creating class
          loader for: " + targetURL );
    java.net.URL[] urls = { new java.net.URL
          ( targetURL ) };
    ClassLoader newClassLoader = new
          java.net.URLClassLoader( urls );
    Thread.currentThread()
          .setContextClassLoader
          ( newClassLoader );
```

```
      // Step 2: load the class and create an
    instance of it.
    System.out.println( "Loading: " +
          targetClass );
    Class urlClass =
          newClassLoader.loadClass
          ( targetClass );
    Object obj = urlClass.newInstance();
    System.out.println( "Object is: \""
          + obj.toString() + "\"" );

    // Step 3: check the URL of the
    loaded class.
    java.net.URL url
          = obj.getClass().getResource
          ( "Welcome.class" );
    if ( url != null )
    {
      System.out.println( "URL used: "
            + url.toExternalForm() );
    }
  }
}
```

When I compile and run this program it produces the folllowing output:

```
Creating class loader for:
http://www.howzatt.demon.co.uk/
Loading: articles.java.Welcome
Object is: "Welcome from Roger Orr's Web site"
URL used:
http://www.howzatt.demon.co.uk/articles/java/W
elcome.class
```

The `URLClassLoader` class supplied with standard Java is doing all the hard work. Obviously there is more to write for a complete solution, for example a `SecurityManager` object may be required in order to provide control over the access rights of the loaded code.

The source code for the 'Welcome.class' looks like this:

```
package articles.java;

public class Welcome
{
  private WelcomeImpl impl
        = new WelcomeImpl();

  public String toString()
  {
    return impl.toString();
  }
}
```

Notice that the class has a dependency upon `WelcomeImpl` - but we did not have to load it ourselves. The same class loader `newClassLoader` we use to load `Welcome` is used by the system to resolve references to dependent classes, and so the system automatically loaded `WelcomeImpl` from the Web site as it was not found locally. There is little code needed for this example and 'it just works' as expected.

Writing your own Class Loader

Although undoubtedly useful the `URLClassLoader` does not provide everything and there will be cases where a new class loader must be written. This might be because you wish to provide a non-standard way of reading the bytes code or to give additional control over the security of the loaded classes. All you need to do is to override the `findClass` method in the new class loader to try and locate the byte code for the named class; the implementation of other methods in `ClassLoader` does not usually need overriding.

Here is a simple example of a class loader which looks for class files with the `.clazz` extension by providing a `findClass` method. This automatically produces a class loader that implements the delegation pattern - the new class loader is only used when the parent class loader is not able to find the class. At this point the `findClass` method shown below is invoked and the `myDataLoad` method tries to obtain the class data from a .clazz file. Although only an example it does illustrate the principles of writing a simple class loader of your own.

```
import java.io.*;
public class MyClassLoader extends ClassLoader
{
  public MyClassLoader( ClassLoader parent )
  {
    super( parent );
  }
  protected Class findClass(String name)
                throws ClassNotFoundException
  {
    try
    {
      byte[] classData = myDataLoad( name );
      return defineClass( name, classData,
            0, classData.length );
    }
    catch ( Exception ex )
    {
      throw new ClassNotFoundException();
    }
  }
  // Example: look for byte code in files
  with .clazz extension
  private byte[] myDataLoad
        ( String name ) throws Exception
  {
    ByteArrayOutputStream bos
          = new ByteArrayOutputStream();
    InputStream is =
          getClass().getResourceAsStream
          ( name + ".clazz" );
    if ( is != null )
    {
      int nextByte;
      while ( ( nextByte = is.read() ) != -1 )
      {
        bos.write( (byte) nextByte );
      }
    }
    return bos.toByteArray();
  }
}
```

We might want to get the Java runtime to install the class loader when the application starts. This can be done by defining the java.system.class.loader property - for this JVM instance - as the class name of our class loader. An object of this class will be constructed at startup, with the 'parent' class loader being the default system class loader. The supplied class loader is then used as the system class loader for the duration of the application.

For example:

```
C:>javac Hello.java
C:>rename Hello.class Hello.clazz
C:>java Hello
Exception in thread "main" java.lang.NoClass
DefFoundError: Hello

C:>java -Djava.system.class.loader=MyClass
LoaderHello

Hello World
```

Class Loading - Java's Answer to DLL Hell?

In practice, for both applets and Web servers, everything does not always work without problem. Unfortunately from time to time there are interactions between the various class loaders and, in my experience, these are typically rather hard to track down.The sort of problems I have had include:

- strange runtime errors caused by different versions of the same class file(s) in different places in the `CLASSPATH`.
- problems with `log4j` generating `NoClassDefFound` or `ClassCastException` errors.
- difficulties registering protocol handlers inside a `WAR` file.

My experience is that resolving these sort of problems is made more difficult by the lack of easy ways to see which class loader was used to load each class in the system. For any given object it is quite easy to track down the class loader - the `getClass()` method returns the correct 'Class' object and calling the `getClassLoader()` method then returns the actual class loader used to instantiate this class. The class loader can be null - for classes loaded by the JVM 'bootstrap' class loader.

Since Java treats any classes loaded by different class loaders as different classes it can be critical to find out the exact class loaders involved. However I do not know of a way to list all classes and their loaders. The Java debugger 'JDB' has a 'classes' command but this simply lists all the classes without, as far as I know, any way to break them down by class loader.

I wanted to find a way to list loaded classes and their corresponding class loader so I could try and identify the root cause of this sort of problem. One way is to extract the source for `ClassLoader.java`, make changes to it to provide additional logging and to place the modified class file in the bootstrap class path before the real `ClassLoader`. This is a technique giving maximum control, but has a couple of downsides. Firstly you need access to the boot class path - this may not always be easy to achieve. Secondly you must ensure the code modified matches the exact version of the JVM being run. After some experimentation, I decided to use reflection on `ClassLoader` itself to provide me pretty well what I wanted.

Reflecting on the Class Loader

Reflection allows a program to query, at run time, the fields and methods of objects and classes in the system. This feature, by no means unique to Java, provides some techniques of particular use for testing and debugging. For example, a test harness such as JUnit can query at run time the methods and arguments of public methods of a target object and then call all methods matching a particular signature. This sort of programming is very flexible, and automatically tracks changes made to the target class as long as they comply with the appropriate conventions for the test harness. However the downside of late binding like this is that errors such as argument type mismatch are no longer caught by the compiler but only at runtime.

There are two main types of reflection supported for a class; the first type provides access to all the public methods and fields for the class and its superclasses, and this is the commonest use of reflection. However there is a second type of reflection giving access to all the declared methods and fields on a class (not including inherited names). This sort of reflection can be used, subject to the security manager granting permission, to provide read (and write) access even to private members of another object.

I noticed that each `ClassLoader` contains a 'classes' Vector that is updated by the JVM for each class loaded by this class loader.

```
[Code from ClassLoader.java in 'java.lang']

// Invoked by the VM to record every loaded
class with this loader.
void addClass(Class c) {
    classes.addElement(c);
}
```

I use reflection to obtain the original vector for each traced class loader and replace it with a proxy object that logs each addition using addElement. The steps are simple, although a lot of work is going on under the covers in the JVM to support this functionality. The class for the ClassLoader itself is queried with the `getDeclaredField` to obtain a 'Field' object for the (private) member 'classes'. This object is then marked as accessible (since by default private fields are not accessible) and finally the field contents are read and written.

The complete code looks something like this:

```
// Add a hook to a class loader (using
reflection)
private void hookClassLoader(
   final ClassLoader currLoader )
{
  try
  {
    java.lang.reflect.Field field =
       ClassLoader.class.getDeclaredField
       ( "classes" );
    field.setAccessible( true );
    final java.util.Vector currClasses =
          (java.util.Vector)field.get
          ( currLoader );
    field.set( currLoader,
          new java.util.Vector() {
       public void addElement( Object o ) {
          showClass( (Class)o );
```

```
            currClasses.addElement(o);
          }
        });
      }
      catch ( java.lang.Exception ex )
      {
        streamer.println( "Can't hook " +
              currLoader + ": " + ex );
        }
      }
```

The end result of running this code against a class loader is that every time the JVM marks the class loader as having loaded a class the `showClass` method will be called. In this method we can take any action we choose based on the newly loaded class. This could be to simply log the class and its loader, or something more advanced.

When I first used reflection to modify the behaviour of a class in Java like this I was a little surprised - I've done similar tricks in C++ but it involves self-modifying code and assembly instructions.

Limitations of this Approach

There are several problems with this approach.

- First of all, it requires sufficient security permissions to be able to access the private member of `ClassLoader`. This is not usually a problem for stand-alone applications but will cause difficulty for applets since the container by default installs a security manager that prevents applet code from having access to the `ClassLoader` fields.
- Secondly, the code is not future proof since it relies upon the behaviour of a private member variable. This does not worry me greatly in this code as it is solely designed to assist in debugging a problem and is not intended to be part of a released program, but some care does need to be taken. What I have done by replacing private member data with a proxy breaks encapsulation.
- Thirdly, the technique is not generally applicable since there must be a suitable member variable in the target class - in this case I was able to override `Vector.addElement()`.
- Fourthly, the code needs calling for each class loader in the system - but there is no standard way for us to locate them all!
- Fifthly, the bootstrap class loader is not included in this code since it is part of the JVM and does not have a corresponding `ClassLoader` object.

It is possible to partly work around the fourth and fifth problems by registering our own class loader at the head of the chain of class loaders. Remember that each class loader in the system (apart from the JVM's own class loader) has a 'parent' class loader. I use reflection to insert my own class loader as the topmost parent for all class loaders.

Once again I achieve my end by modifying a private member variable of the classloader - this time the 'parent' field.

```
/**
 * This method injects a ClassLoadTracer
 object into the current class loader chain.
 * @param parent the current active class
   loader
 * @return the new (or existing) tracer object
 */
public static synchronized ClassLoadTracer
   inject( ClassLoader parent )
{
  // get the current topmost class loader.
  ClassLoader root = parent;
  while ( root.getParent() != null )
     root = root.getParent();
  if ( root instanceof ClassLoadTracer )
     return (ClassLoadTracer)root;
  ClassLoadTracer newRoot = new
     ClassLoadTracer( parent );
  // reflect on the topmost classloader to
     install the ClassLoadTracer ...
  try
  {
    // we want root->parent = newRoot;
    java.lang.reflect.Field field =
       ClassLoader.class.getDeclaredField(
       "parent" );
    field.setAccessible( true );
    field.set( root, newRoot );
  }
  catch ( Exception ex )
  {
    ex.printStackTrace();
    System.out.println( "Could not install
       ClassLoadTracer: " + ex );
  }
  return newRoot;
}
```

The end result of calling the above method against an existing class loader is that the top-most parent becomes an instance of my own `ClassLoadTracer` class. This class, yet another extension of ClassLoader, overrides the `loadClass` method to log successful calls to the bootstrap class loader (thus solving the fifth problem listed above). It also keeps track of the current thread context class loader to detect any class loaders added to the system and thus helps to resolve the fourth problem.

Note however that this is only a partial solution since there is no requirement that class loaders will follow the delegation technique and so it is possible that my `ClassLoadTracer` will never be invoked. However, for the cases I have used it the mechanism seems to work well enough for me to get a log of the classes being loaded by the various class loaders.

Conclusion

Class loaders are powerful, but there does not seem to be enough debugging information supplied as standard to resolve problems when the mechanism breaks down. I have shown a couple of uses of reflection to enable additional tracing to be provided where such problems exist. The techniques shown are of wider use too, enabling some quite flexible debugging techniques that add and remove probes from target objects in the application at runtime.

All the source code for this article is available at:
`http://www.howzatt.demon.co.uk/articles/ClassLoading.zip`

Thanks are due to Alan Griffiths, Richard Blundell and Phil Bass who reviewed drafts of this article and suggested a number of improvements.

Roger Orr
rogero@howzatt.demon.co.uk

Software Project Management: Adding Stakeholder Metrics to Agile Projects

by Tom Gilb

Abstract. Agile methods need to include stakeholder metrics in order to ensure that projects focus better on the critical requirements, and that projects are better able to measure their achievements, and to adapt to feedback. This paper presents a short, simple defined process for evolutionary project management (Evo), and discusses its key features.

Introduction

In 2001, a British Computer Society Review paper indicated that only 13% of 1027 surveyed IT projects were 'successful' (Taylor 2001). In the same year, a Standish report indicated that although there has been some recent improvement, 23% of their surveyed projects were considered total failures and only 28% totally successful (that is, on time, within budget and with all the required functionality) (Johnson 2001: Extracts from Extreme Chaos 2001, a Standish Report). The US Department of Defense, a few years ago, estimated that about half its software projects failed (Personal Communication, Norm Brown, SPMN (Software Program Managers Network)/Navy). While these figures represent an improvement on the 50% reported for failed DoD projects when the Waterfall Method dominated (Jarzombek 1999), they are still of extreme concern. We must be doing something very wrong. What can senior management and IT project management do about this situation in practice?

Some people recommend complex development process standards such as CMM (Capability Maturity Model®), CMMI (Capability Maturity Model® Integration), SPICE (Software Process Improvement and Capability dEtermination) and their like. I am not convinced that these are 'good medicine' for even very large systems engineering projects, and certainly they are overly complex for most IT projects.

Other people recommend agile methods – these are closer to my heart – but maybe, for non-trivial projects they are currently 'too simple'?

Stakeholder Metrics

I believe agile methods would benefit if they included 'stakeholder metrics'. I have three main reasons for suggesting this:

- to focus on the critical requirements: All projects, even agile projects, need to identify all their stakeholders, and then identify and focus on the 'top few' critical stakeholder requirements.
- to measure progress: Critical requirements need to be quantified and measurable in practice. Quantified management is a

A Simple Evolutionary Project Management Method

Tag: Quantified Simple Evo Project. Version: July 8, 2003. Owner: Tom@Gilb.com.
Status: Draft.

Project Process Description

1. Gather from all the key stakeholders the top few (5 to 20) most critical performance goals (including qualities and savings) that the project needs to deliver. Give each goal a reference name (a tag).
2. For each goal, define a scale of measurement and a 'final target' goal level. For example, *Reliability: Scale: Mean Time Between Failure, Goal: >1 month.*
3. Define approximately 4 budgets for your most limited resources (for example, time, people, money, and equipment).
4. Write up these plans for the goals and budgets (*Try to ensure this is kept to only one page*).
5. Negotiate with the key stakeholders to formally agree these goals and budgets.
6. Plan to deliver some benefit (that is, progress towards the goals) in *weekly* (or shorter) cycles (Evo steps).
7. Implement the project in Evo steps. Report to project sponsors after each Evo step (weekly, or shorter) with your best available estimates or measures, for each performance goal and each resource budget.
   - O*n a single page,* summarize the *progress to date* towards achieving the goals and the costs incurred.
   - Based on numeric feedback, and stakeholder feedback; *change whatever needs to be changed to reach the goals within the budgets.*
8. When all goals are reached: "Claim success and move on" (Gerstner 2002). Free the remaining resources for more profitable ventures

Project Policy

1. The project manager, and the project, will be judged exclusively on the relationship of progress towards achieving the goals versus the amounts of the budgets used. The project team will do anything legal and ethical to deliver the goal levels within the budgets.
2. The team will be paid and rewarded for 'benefits delivered' in relation to cost.
3. The team will find their own work process, and their own design.
4. As experience dictates, the team will be free to suggest to the project sponsors (stakeholders) adjustments to the goals and budgets to 'more realistic levels.'

Figure 1: A recommended Project Management Method

necessary minimum to control all but the smallest upgrade efforts.

- to enable response to feedback: By responding to real experience and modifying plans accordingly, projects can make better progress. This is something that agile projects with their short cycles can especially utilize.

In this paper, I shall present a simple, updated 'agile', evolutionary project management process and explain the benefits of a more focused, quantified approach.
I recommend the evolutionary project management process and policy shown in Figure 1.

This simple project process and policy captures all the key features: you need read no more! However, in case any reader would like more detail, I will comment on the process and policy definition, statement by statement, in the remainder of this paper.

Project Process Description

1. **Gather from all the key stakeholders the top few (5 to 20) most critical goals that the project needs to deliver.**

   Projects need to learn to focus on all the stakeholders that arguably can affect the success or failure. The needs of all these stakeholders must be determined – by any useful methods – and converted into project requirements. By contrast, the typical agile model focuses on a user/customer 'in the next room'. Good enough if they were the only stakeholder, but disastrous for most real projects, where the critical stakeholders are more varied in type and number. Agile processes, due to this dangerously narrow requirements focus, risk outright failure, even if the one identified 'customer' gets all their needs fulfilled.

2. **For each goal, define a scale of measurement and a 'final target' goal level. For example, Reliability: Scale: Mean Time Before Failure, Goal: >1 month.**

   Using Evo, a project is initially defined in terms of clearly stated, quantified, critical objectives. Agile methods do not have any such quantification concept. The problem is that vague targets with no quantification and lacking in deadlines do not count as true goals: they are not measurable, and not testable ideas.

   Note that in Evo the requirements are not cast in concrete, even though they are extremely specific. During a project, the requirements can be changed and tuned based on practical experience, insights gained, external pressures, and feedback from each Evo step (See also point 4 under 'Project Policy').

3. **Define approximately 4 budgets for your most limited resources (for example, time, people, money, and equipment).**

   Conventional methods do set financial and staffing budgets, but usually at too macro a level. They do not seem to directly, and in detail, manage the array of limited resources we have. Admittedly there are some such mechanisms in place in agile methods, such as the incremental weekly (or so) development cycle (which handles time). However, the Evo method sets an explicit numeric budget for any useful set of limited resources, effort, calendar time, financial spend, or memory space.

4. **Write up these plans for the goals and budgets (Ensure this is kept to only one page).**

   All the key quantified performance targets and resource budgets should be presented simultaneously on a single overview page. Additional detail about them can, of course, be captured in additional notes, but not on this one 'focus' page.

5. **Negotiate with the key stakeholders to formally agree these goals and budgets.**

   Once the requirements, derived from the project's understanding of the stakeholder needs, are clearly articulated, we need to go back to the real stakeholders and check that they agree with our 'clear' (but potentially incorrect or outdated) interpretation.

   It is also certainly a wise precaution to check back later, during the project evolution, with the stakeholders, especially the specific stakeholders who will be impacted by the next Evo step:

   - as to how they feel about a particular choice of step content (that is, how they see the proposed design impacting performance and cost, and whether the original impact estimates are realistic in the real current implementation environment, and
   - to check for any new insights regarding the long term requirements.

6. **Plan to deliver some benefit (that is, 'progress towards the goals') in weekly (or shorter) cycles (Evo steps).**

   A weekly delivery cycle is adopted by agile methods; this is good. However, the notion of measurement each cycle, on multiple performance and resource requirements, is absent.
   Using Evo, the choice of the next Evo step is based on highest stakeholder value to cost ratios. It is not simply, "What shall we do next?" It is "What is most effective to do next - of highest value to the stakeholders with consideration of resources?"

   The agile methods' notion of agreeing with a user about the function to be built during that weekly cycle is healthy, but the Evo method is focused on systematic, weekly, measured delivery towards long-range higher-level objectives, within numeric, multiple, resource constraints. This means that the Evo method is more clearly focused on the wider stakeholder-set values, and on total resource cost management.

   The Evo method is *not focused* on simply writing code ('we are programmers, therefore we write code'). The Evo method is focused on delivering useful results to an organically whole system. We reuse, buy or exploit existing code just as happily as writing our own code. We build databases, train and motivate users, improve hardware, update telecommunications, create websites, improve the users' working environment, and/or improve motivation. So we become more like systems engineers ('any technology to deliver the results!'), than programmers ('what can we code for you today?').

   Table 1 shows the use of an Impact Estimation table (Gilb 2004) to plan and track critical performance and cost characteristics of a system (Illustration courtesy of Kai Gilb). The pair of numbers in the three left hand columns (30, 5 etc.) show initial benchmarks (30, 99, 2500) and Goal levels (5, 200, 100,000). The '%' figures are the real scale impacts (like 20) converted to a % of the way from benchmark to the Goal levels (like 20% of the distance from benchmark to Goal).

	Step 12				Step 13			
	Buttons.Rubber				Buttons.Shape and Layout			
	Estimate		Actual		Estimate		Actual	
Goals User-Friendliness.Learn 30 min<->5 min by one year	-10	33%	-5	17%	-5	20%	5	-20%
Reliability 99 days <-> 200 days by one year	-3	-3%	-1	-1%	20	20%	2	2%
Budgets Project-Budget 25000 • <-> 1000000 • by one year	2000	2%	2500	3%	1000	1%	1000	1%

Table 1: An Impact Estimation Table

7. Implement the project in Evo steps and report your progress after each Evo step.

Report to the project sponsors after each Evo step (weekly, or shorter) with your best available estimates or measures, for each performance goal and each resource budget.

- On a single page, summarize the progress to date towards achieving the goals and the costs incurred.
- Based on the numeric feedback and stakeholder feedback, change whatever needs to be changed to reach the goals within the budgets.

All agile methods agree that the development needs to be done in short, frequent, delivery cycles. However, the Evo method specifically insists that the closed loop control of each cycle is done by:

- making numeric pre-cycle estimates,
- carrying out end-cycle measurements,
- analyzing deviation of measurements from estimates,
- making appropriate changes to the next immediate planned cycles,
- updating estimates as feedback is obtained and/or changes are made,
- managing stakeholder expectations ('this is going to late, if we don't do X').

The clear intention to react to the feedback from the metrics and to react to any changes in stakeholder requirements is a major feature of Evo. It helps ensure the project is kept 'on track' and it ensures relevance. It is only by the use of stakeholder metrics that Evo is allowed to have such control.

Figure 2 shows results being cumulated numerically step by step until the Goal level is reached. In a UK Radar system (Author experience), the system was delivered by gradually building database info about planes and ships, tuning recognition logic, and tuning the radar hardware.

8. When all the goals are reached: 'Claim success and move on' (Gerstner 2002). Free remaining resources for more profitable ventures.

A major advantage of using numeric goal and budget levels, compared to 'a stream of yellow stickies from users' (a reference to agile method practice), is that it is quite clear when your goals are reached within your budgets. In fact, 'success' is formally well defined in advance by the set of the required numeric goal and budget levels.

Projects need to be evaluated on 'performance delivered' in relation to 'resources used'. This is a measure of project management 'efficiency'. When goals are reached, we need to avoid misusing resources to deliver more than is required. No additional effort should be expended to improve upon a goal, unless a new improved target level is set.

Project Policy

1. The project manager, and the project, will be judged exclusively on the relationship of progress towards achieving the goals versus the amounts of the budgets used. The project team will do anything legal and ethical to deliver the goal levels within the budgets.

Projects need to be judged primarily on their ability to meet critical performance characteristics, in a timely and profitable way. This cannot be expected if the project team is paid 'by effort expended'.

2. The team will be paid and rewarded for benefits delivered in relation to cost.

Teams need to be paid according to their project efficiency, that is by the results they deliver with regard to the costs they incur. Even if this means that super efficient teams get terribly rich! And teams that fail go 'bankrupt.'

When only 13% of 1027 IT projects are 'successful' (Taylor 2001), we clearly need to find better mechanisms for rewarding success, and for not rewarding failure. I suggest that sharp numeric definition of success levels and consequent rewards for reaching them, is minimum appropriate behavior for any software project.

3. The team will find their own work process and their own design.

Agile methods believe we need to reduce unnecessarily cumbersome corporate mandated processes. I agree. They also believe in empowering the project team to find the processes, designs and methods that really work for them locally. I heartily agree!

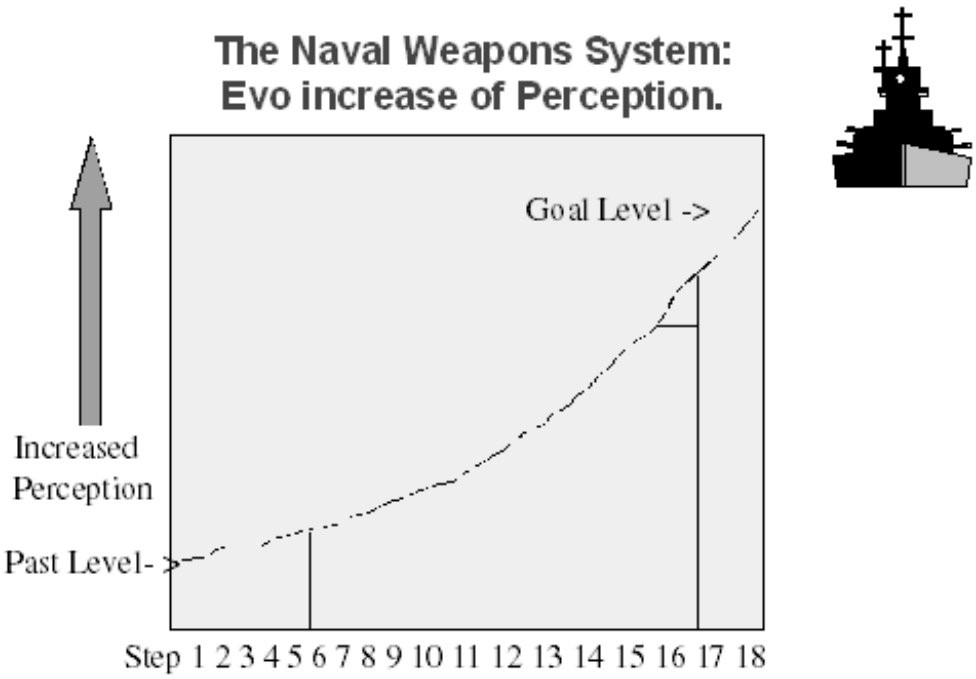

Figure 2: Cumulation of results towards goal level

However, I also strongly believe that numeric definition of goals and budgets, coupled with frequent estimation and measurement of progress, are much-needed additional mechanisms for enabling this empowerment. The price to pay for this, a few estimates and measures weekly, seems small compared to the benefits of superior control over project efficiency.

4. **As experience dictates, the team will be free to suggest to the project 'sponsors' (one type of stakeholder) adjustments to 'more realistic levels' of the goals and budgets.**

   No project team should be 'stuck' with trying to satisfy unrealistic or conflicting stakeholder dreams within constrained resources.

   Further, a project team can only be charged with delivering inside the 'state of the art' performance levels at inside the 'state of the art' costs. Exceeding 'state of the art' performance is likely to incur 'exponential' costs.

Summary

A number of agile methods have appeared, trying to simplify project management and systems implementation. They have all missed two central, fundamental points; namely quantification and feedback.

Evolutionary project management (Evo) uses quantified feedback about critical goals and budgets. Further, Evo also insists that early, frequent, small, high stakeholder value deliveries (Evo steps) are made to real users: this is only possible if supported by stakeholder metrics.

It is the use of stakeholder metrics that allows better focus, more measurement of progress, and more flexibility to change. It is time agile methods adopted quantified, critical stakeholder metrics.

Tom Gilb
www.gilb.com

References

Abrahamsson, Pekka, Outi Salo, Jussi Ronkainen and Juhani Warsta, *Agile Software Development Methods. Review and Analysis*, VTT Publications, Espoo, Finland, 2002, ISBN 951-38-6009-4, URL: www.inf.vtt.fi/pdf/, 107 pp.

Gerstner, Louis V. Jr., *Who Says Elephants Can't Dance? Inside IBM's Historic Turnaround,* HarperCollins, 2002, ISBN 0007153538.

Gilb, Tom, *Principles of Software Engineering Management*, Addison-Wesley, 1988, ISBN 0201192462.

Gilb, Tom, *Competitive Engineering: A Handbook for Systems & Software Engineering Management using Planguage*, See www.Gilb.com for draft manuscript, 2004.

Jarzombek, S., *Proceedings of the Joint Aerospace Weapons Systems Support*, Sensors and Simulation Symposium, Government Printing Office Press, 1999. Source Larman & Basili 2003.

Johnson, Jim, Karen D. Boucher, Kyle Connors, and James Robinson, "*Collaborating on Project Success*," Software Magazine, February 2001.www.softwaremag.com/L.cfm?Doc=archive/2001feb/CollaborativeMgt.html

Johnson, Jim, "*Turning Chaos into Success*," Software Magazine, December 1999. www.softwaremag.com/L.cfm?Doc=archive/1999dec/Success.html

Larman, Craig, *Agile and Iterative Development: A Manager's Guide*, Addison Wesley, 2003.

Larman, Craig, and Victor Basili, "*Iterative and Incremental Development: A Brief History*," IEEE Computer, June 2003, pp 2-11.

Taylor, Andrew, "*IT projects sink or swim*," BCS Review, 2001. http://www.bcs.org.uk/review/2001/articles/itservices/projects.htm

C-side Re-sort

by Kevlin Henney

Testing is one way of making the essentially invisible act of software development more visible [7]. Testing can occur at many different levels of a system's architecture, from whole system down to individual code modules, and in many different stages of a system's development. In the context of modern agile software development, testing is normally associated with Test-Driven Development (TDD), an approach that subsumes conventional unit-level testing, complementing its traditionally quantitative role with a qualitative practice of design.

In spite of the attention given to object-oriented development, TDD and modern testing frameworks, it is worth understanding how and why unit testing has an important role to play in general, regardless of the technologies or broader development philosophy involved. This understanding also applies to TDD when considered solely in terms of its testing aspect.

This article walks through a simple example, highlighting some of the motivation behind basic unit testing and the practices involved [8] but without following through into TDD. Thus it is a programmer testing responsibility to carry out code-centric tests, but automated tests ensure that unit-level tests are executed as code on code rather than by programmer on code. Example-based test cases, as opposed to exhaustive tests, ensure that test cases adopt a sustainable black-box approach to testing.

Standard portable C is used in the narrative example to remind and emphasise that the applicability of unit-testing techniques is not restricted to object-oriented programming. It also demonstrates that, at the very minimum, nothing more than the humble assert is needed to realise test cases. The narrative also highlights the relationship between the contract metaphor and testing, and the tension between programming by contract as an assertion style and the use of contracts to inform test cases. Thus the story also helps to clarify some of the common misunderstandings concerning the relationship between programming by contract and testing.

A Sort of Beginning

Consider the following situation: a cross-platform C program whose logic involves, amongst other things, the sorting of a large number of integers; furthermore, sorting is found to dominate the performance and needs to be optimised.

The sorting function used is the C standard library qsort function. In spite of its name, it is not required to be implemented as quicksort. However, it is not the time complexity that is considered to be the issue — the implementations on the supported platforms appear to be fairly optimal in this respect — but the cost of comparison. The qsort function is generic in the sense that it works with untyped chunks of memory, i.e. void * to pass an array and size_t to indicate element count and size, but this means that the comparison must be passed in as a pointer to a function that performs the comparison correctly based on the actual types:

```
int int_comparison(const void *lhs_ptr,
                   const void *rhs_ptr)
{
  int lhs = *((const int *) lhs_ptr);
  int rhs = *((const int *) rhs_ptr);
  return lhs < rhs ? -1 : lhs == rhs ? 0 : 1;
}
```

This function is based on strcmp semantics, so that the result is less than zero, equal to zero or greater than zero depending on whether the left-hand argument compares less than, equal to or greater than the right-hand argument. The cost of continually calling a function through a pointer to perform what is essentially the single machine instruction for comparing integers is what is incurring the overhead.

The Programmer's Responsibility

The favoured solution is to replace the use of `qsort` with a custom-written function — let's call it `quickersort` — that is written directly in terms of `int`:

```
void quickersort(int base[], size_t length);
```

It is not necessarily hard to find a good algorithm — there is no shortage of published examples available — but who is responsible for ensuring that the function is correctly implemented? It is supposed to be a drop-in replacement and an improvement: a function that does not work or whose performance is worse than with qsort is unlikely to impress.

Of course, it is a programmer's responsibility to be diligent in implementation, but care in coding is not enough: sorting algorithms are notorious for defects that arise from typos, thinkos, assumptions and oversights; second guessing a would-be optimisation's effectiveness is often a fool's game; someone else may know a more effective solution. Peer review and static analysis of code can help — and should be employed — but the ultimate proof of the pudding is in the eating: the code needs to be tested.

But with whom does the responsibility for testing lie? Ultimately a test of the software as a whole will pick up many problems, but this system view is somewhat second hand, removed from the point of the change that has been so precisely specified. Given that system-level testing is often a responsibility separated from that of code development, leaving problems to be discovered so indirectly introduces long feedback loops and a greater element of chance into the process of development. And given the precise nature of the requirement — to speed up sorting — and of the change — replace qsort with a hand-crafted alternative that is functionally equivalent but faster — to hand off a slower, defective piece of code as a fait accompli can be considered less than professionally responsible.

It is therefore a programmer testing responsibility to ensure a basic level of confidence in the code. Programmers are in the best position to catch a large class of possible problems before they introduce friction into the development process by causing others and themselves further problems. However, no programmer is perfect and programmers do not have unlimited time to spend crafting and checking their code. Some defects may remain, appearing only in a larger, more integrated context. This system perspective can be considered a separate responsibility, owned and exercised by others.

Test Automation

Returning to the programmer perspective, the next question is what to test for and how? Perhaps the most obvious feature of the requirement, as related to the code, is that quickersort should run faster than qsort. To do this effectively requires more data than can be typed in conveniently at a command-line driven test

overload issue 68 august 2005

harness. Other than being arbitrary, the specific values don't matter, so a function can be used to populate an array of given length at runtime:

```
int *fill_unsorted(int base[], size_t length);
// returns base
```

A simple implementation can be written in terms of the standard rand function, but alternative, more application-specific distributions may be appropriate. Given this, it is possible to write a simple function to compare qsort and quickersort against one another on the same arbitrary data set:

```
void a_test_of_sorts(size_t length)
{
  size_t byte_length = length * sizeof(int);
  int *for_qsort     = fill_unsorted((int *)
      malloc(byte_length), length);
  int *for_quickersort = (int *)
      memcpy(malloc(byte_length), for_qsort,
      byte_length);
  clock_t start;

  start = clock();
  qsort(for_qsort, length, sizeof(int),
      int_comparison);
  print_result("qsort", length,
      clock() - start);

  start = clock();
  quickersort(for_quickersort, length);
  print_result("quickersort", length,
      clock() - start);

  free(for_quickersort);
  free(for_qsort);
}
```

Arrays of the specified length are allocated and populated with the same data set, and then the times for qsort and for quickersort are taken and reported. The following reporting function presents results in comma-separated format with times in milliseconds:

```
void print_result(const char *label,
      size_t length, clock_t elapsed)
{
  printf("%i, %i, %s\n", (int) length,
      (int)(elapsed * 1000 / CLOCKS_PER_SEC),
      label);
}
```

This simple infrastructure allows simple and repeatable automated tests. The code above can be wrapped in an outer loop driven either from outside the C code in a script or from within main. The outer loop can step through different array lengths, generating results that can be piped into a file for analysis. This performance test can be included for automatic execution as part of the check-in process or the integration build.

A Contractual Dead End

However, having demonstrated that quickersort is indeed quicker than qsort, the programmer still has unfinished business. Mindful of producing an algorithm that is fast but wrong, the programmer needs to check that the resulting array satisfies the functional contract for sorting. In this case the functional contract can be phrased in terms of pre- and postconditions, but it is perhaps more subtle than many might first assume [6]:

- *precondition*: base is a valid non-null pointer to the initial element of an array of initialised ints at least length elements long.
- *postcondition*: The values in the array defined by base and length are sorted in ascending order, with equal values adjacent to one another, and they are a permutation of the values in the array before it was sorted.

It is not uncommon for programmers to assume that sorting operations have no precondition and have only the requirement for the array to be sorted as the postcondition.

Given a contract phrased in assertive form, there is a common assumption that the "right thing to do" with it is to place corresponding assertions within the production code and check the pre- and postcondition directly. It is worth noting that, with the exception of base being non-null, the truth or falsehood of the precondition cannot be established within a function written in standard C. It is also worth noting that attempting to check the postcondition in the function has some unfortunate consequences:

```
void quickersort(int base[], size_t length)
{
  #ifndef NDEBUG
  size_t byte_length = length * sizeof(int);
  int *original = (int *) malloc(byte_length);
  assert(base);
  memcpy(original, base, byte_length);
  #endif

  ...// sorting algorithm of choice
  #ifndef NDEBUG
  assert(is_sorted(base, length));
  assert(is_permutation(base, original,
        length));
  free(original);
  #endif
}
```

Perhaps the most visible effect is the reduced readability of the code: the plumbing needed to support the postcondition check is anything but discreet. The infrastructure code is also error prone and somewhat incomplete: although undoubtedly useful, `is_sorted` and `is_permutation` are not standard functions. They must be written by the programmer especially for this task... and they need to be tested. They are sufficiently complex that to be sure of their correctness more than a walkthrough is needed: `is_sorted` is not so much of a challenge, although it is easy to introduce off-by-one errors; an effective version of `is_permutation` is more demanding, even with the simplifying assumption that base is already sorted; getting either of them

wrong leads to wasted time and effort spent determining whether it is the `quickersort` algorithm or the implementation of the postcondition check that is incorrect — or, in the worst case, both.

The resulting complexity rivals and typically exceeds that of the actual `quickersort` implementation. Indeed, this observation can be taken quite literally: it is not only the complexity of the code, the coding and the testing that have risen; it is also the operational complexity. The complexity of `is_sorted` is *O(n)* but, unless more heap memory is used, that of `is_permutation` is *O(n2)*. It is unlikely that the programmer, whose goal is to produce a faster sorting operation, would be happy to decelerate the performance to *O(n2)*. Even with an O(*n* log *n*) implementation of `is_permutation`, the effect of the extra algorithms and the additional dynamic memory usage will trounce any performance advantage that `quickersort` might otherwise have delivered.

This issue serves to highlight the difference between production and debugging builds. By default, the standard assert macro is enabled, but when `NDEBUG` is defined it compiles to a no-op, hence the use of the `NDEBUG` conditionally compiled code in the previous sketch of a self-checking quickersort. A release build should ensure that `NDEBUG` is defined for compilation; any bugs in the implementation of quickersort can be trapped only in non-release builds.

There is also duplicity in this attempt to writing self-checking code. Conditional compilation often causes more problems than it resolves [12]. In this case it leads to a style of development perhaps best described as *debug by contract*. The operational test, `a_test_of_sorts`, can only be run meaningfully with a release build of `quickersort`, so any faults cannot be detected by this active use of quickersort. A released version of the whole program will also not detect any defects, so any defects that remain in the final version of `quickersort` will emerge only as incorrect results at runtime after the program has been deployed — assuming that its users notice.

What about system-level testing? Those responsible for testing the system as a whole — whether another programmer on the same the team, a dedicated tester on the same team or a separate testing team — will have to be supplied with two builds of the system: a release version and a debug-enabled version with different operational behaviour, but (one hopes) identical functional behaviour. This is more often a recipe for extra work than it is one for assuring quality. Assuming that system regression testing misses something as fundamental as incorrectly sorted data, if the debug-enabled version trips an assertion in a faulty quickersort, the tester must now feed the problem back to the programmer to fix... wait for the fix... and then proceed. This longer feedback cycle was precisely one of the issues supposed to be addressed by introducing programmer testing responsibility. The code is fast but wrong, and wrong in a way that could have been easily detected by the programmer.

Here's One I Prepared Earlier

What is lacking is some basic sign-off on the functionality of the code. The programmer could poke values in at a simple command line test harness to see whether they were sorted correctly or not. Or the programmer could use automated tests.

There are two refactoring steps that provide a (very) basic reality check on the functional behaviour.

The first is to do away with the need for `is_sorted` and `is_permutation`:

```
void quickersort(int base[], size_t length)
{
  #ifndef NDEBUG
  size_t byte_length = length * sizeof(int);
  int *sorted        = (int *)
      malloc(byte_length);
  assert(base);
  memcpy(sorted, base, byte_length);
  qsort(sorted, length, sizeof(int),
      int_comparison);
  #endif

  ... // sorting algorithm of choice
  #ifndef NDEBUG
  assert(memcmp(base, sorted, byte_length)
        == 0);
  free(original);
  #endif
}
```

In other words, compare the result against the expected result produced by another means — a luxury that is not always available, but is on hand in this particular example. The next step is to recognise and capitalise on the similarity between this new assertion code and the functionality of `a_test_of_sorts`:

```
void a_test_of_sorts(size_t length)
{
  ...
  assert(memcmp(for_quickersort, for_qsort,
      byte_length) == 0);
  free(for_quickersort);
  free(for_qsort);
}
```

For the scenario defined by `a_test_of_sorts`, the introduction of this single `assert` and the elimination of all the infrastructure in `quickersort` is a great simplification that achieves the same effect as before. The assertion for a non-null base in `quickersort` can be retained, but the benefits of keeping it appear somewhat marginal — most modern operating systems will trap incorrect uses of null pointers and, unless an operation's implementation actually requires it, it is entitled to weaken its precondition and therefore accommodate a null.

Test by Example

However, although the performance-related automated tests now have the additional effect of checking the functional correctness of their results — and at no extra cost — the programmer can only confirm that `quickersort` passes for large sets of randomly generated data. Certain scenarios and boundary cases may or may not have been checked as a result. Without checking or rigging the generated data sets, the programmer cannot be sure that `a_test_of_sorts` is providing suitable coverage of these.

The programmer can make these situations explicit by writing example-based test cases that check certain situations with specific values. For example, the following test ensures that sorting an empty array does not contain any off-by-one memory accesses:

```
int empty[] = { 2, 1 };
quickersort(empty + 1, 0);
assert(empty[0] == 2);
assert(empty[1] == 1);
```

In a similar vein, the following test ensures the same for sorting a single-element array and ensures that the single-element's value remains unchanged:

```
int single[] = { 3, 2, 1 };
quickersort(single + 1, 1);
assert(single[0] == 3);
assert(single[1] == 2);
assert(single[2] == 1);
```

The following ensures that sorting an array of identical elements is an identity operation:

```
int identical[] = { 3, 2, 2, 2, 1 };
quickersort(identical + 1, 3);
assert(identical[0] == 3);
assert(identical[1] == 2);
assert(identical[2] == 2);
assert(identical[3] == 2);
assert(identical[4] == 1);
```

And likewise for an ascending sequence of elements:

```
int ascending[] = { 2, -1, 0, 1, -2 };
quickersort(ascending + 1, 3);
assert(ascending[0] == 2);
assert(ascending[1] == -1);
assert(ascending[2] == 0);
assert(ascending[3] == 1);
assert(ascending[4] == -2);
```

Of course, it is worth testing that `quickersort` actually does something, given that all the previous test cases could be met successfully with an empty implementation! Here is a test on a reverse-sorted array:

```
int descending[] = { 3, 2, 1, 0, -1 };
quickersort(descending + 1, 3);
assert(descending[0] == 3);
assert(descending[1] == 0);
assert(descending[2] == 1);
assert(descending[3] == 2);
assert(descending[4] == -1);
```

Further tests, based on other sample data characteristics, can be added in this style: ascending with some adjacent identical values, arbitrarily ordered distinct values, arbitrary values including some duplicates, arbitrary values including `INT_MIN`, etc.

At this point programmer testing responsibility can be said to have been reasonably fulfilled: both operational and functional aspects of `quickersort` are checked in automated tests, with example-based test cases providing basic unit-testing coverage of the functionality.

Sortie

The mistaken notion that testing is someone else's problem is unfortunately quite common, regardless of development process [10]:

> *Programmers today aren't sure their code is bug-free because they've relinquished responsibility for thoroughly testing it. It's not that management ever came out and said, "Don't worry about testing your code—the testers will do that for you." It's more subtle than that. Management expects programmers to test their code, but they expect testers to be more thorough; after all, that's Testing's full-time job.*

The notion that testing is part of programming and not something foreign can be seen to have support both from perspectives that support agile development [4]:

> *Responsibilities of developers include understanding requirements, reviewing the solution structure algorithm with peers, building the implementation, and performing unit testing.*

And from dedicated testing practitioners [1]:

> *I find the projects I work on usually go more smoothly when programmers do some unit and component testing of their own code. Through the ascendance of approaches like Extreme Programming, such a position is becoming less controversial. [...] So, a good practice is to adopt a development process that provides for unit testing, where programmers find bugs in their own software, and for component testing, where programmers test each other's software. (This is sometimes called 'code swapping.') Variations on this approach use concepts like pair programming and peer reviews of automated component test stubs or harnesses.*

Systems should be tested at the many levels at which they are conceived, from the system level down to the function level. These different views often suggest a division of responsibility and ownership with respect to different kinds of testing [2]:

> *Unit testing involves testing the individual classes and mechanisms; is the responsibility of the application engineer who implemented the structure. [...] System testing involves testing the system as a whole; is the responsibility of the quality assurance team.*

Programmers inevitably write code and scripts to test parts of their system, but these are often ad hoc fragments scattered around personal directories rather than versioned along with other project artefacts and integrated into system builds. There are good grounds for aiming for a higher level of automation [11]:

> *As long as you've got them, developer and customer tests should be automated as much as possible and run as part of the daily build.*

If the tests are not automated or if they take too much time, they won't be run often enough. Big batches of changes will be made before testing, which will make failure much more likely, and it will be much more difficult to tell which change caused the tests to fail.

We can also appreciate automation from the human as well as the technological perspective [3]:

*Teams do deliver successfully using manual tests, so this can't be considered a critical success factor. However, every programmer I've interviewed who once moved to automated tests swore **never to work without them again**. I find this nothing short of astonishing.*

Their reason has to do with improved quality of life. During the week, they revise sections of code knowing they can quickly check that they hadn't inadvertently broken something along the way. When they get code working on Friday, they go home knowing that they will be able on Monday to detect whether anyone had broken it over the weekend—they simply rerun the tests on Monday morning. The tests give them freedom of movement during the day and peace of mind at night.

Assertions in production code often end up encouraging a code-and-fix mindset rather than a more considered one. A failed production-code assertion in an arbitrary situation gives you something — but not a great deal — to work with. A failed assertion in a specified test case gives you a precise scenario that defines the context for failure.

The relationship between testing and contracts is often misunderstood, and often in a way that creates extra work without offering significant improvement in design quality. There is a far higher probability that a programmer will incorrectly specify a postcondition than a unit test. For example, at the time of writing, the first entry that Google brings up for "design by contract" is an explanation of how to write bug-free software in Eiffel, a language that famously supports pre- and postcondition checking [5]. It can be considered ironic that this paper contains poorly and incorrectly specified pre- and postconditions. Tests would have uncovered these defects immediately.

The specification of a precondition and a postcondition for a sorting operation is subtle but not necessarily hard — just harder than many expect. However, converting those conditions into practical assertions raises a good many questions and generates a great deal of code complexity. By contrast, it is trivial to test such an operation and there is little complexity involved — literally: the example-based test cases in the sorting example have O(1) operational complexity and a McCabe cyclomatic complexity value of 1, whereas the attempt to write a correct postcondition in code involved greater complexity on all fronts. Thus contracts lay down the lines that tests can follow, and vice versa [9]:

We like to think of unit testing as testing against contract. We want to write test cases that ensure that a given unit honors its contract. This will tell us two things: whether the code meets the contract, and whether the contract means what we think it means. We want to test that the module delivers the functionality it promises, over a wide range of test cases and boundary conditions.

And this relationship between contracts and testing provides us with a hint and a useful bridge to expanding the role of testing into design. The example in the sorting story was tightly scoped in terms of interface and implementation, and it was strictly focused on demonstrating the responsibilities and practice of more conventional unit testing. Test-Driven Development takes this a step further, empowering unit testing to support and contribute to design. But that's another story.

Kevlin Henney
kevlin@curbralan.com
kevlin@acm.org

References

[1] Rex Black, *Critical Testing Processes*, Addison-Wesley, 2004.
[2] Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2nd edition, Benjamin/Cummings, 1994.
[3] Alistair Cockburn, *Crystal Clear: A Human-Powered Methodology for Small Teams*, Addison-Wesley, 2005.
[4 James O Coplien and Neil B Harrison, *Organizational Patterns of Agile Software Development*, Prentice Hall, 2005.
[5] "Building bug-free O-O software: An introduction to Design by Contract™", `http://archive.eiffel.com/doc/manuals/technology/contract/`.
[6] Kevlin Henney, "*Sorted*", Application Development Advisor, July 2003, available from `http://www.curbralan.com`.
[7] Kevlin Henney, "*Five Considerations*", keynote at ACCU Spring Conference, April 2005.
[8] Kevlin Henney, "*Driven to Tests*", Application Development Advisor, May 2005.
[9] Andrew Hunt and David Thomas, *The Pragmatic Programmer*, Addison-Wesley, 2000.
[10] Steve Maguire, *Writing Solid Code*, Microsoft Press, 1993.
[11] Mary Poppendieck and Tom Poppendieck, *Lean Software Development*, Addison-Wesley, 2003.
[12] Henry Spencer and Geoff Collyer, "#ifdef Considered Harmful, or Portability Experience with C News", USENIX, June 1992, `http://www.literateprogramming.com/ifdefs.pdf`