The magazine of the ACCU

www.accu.org

{cvu}

Volume 21 Issue 2 May 2009 £3

Features

{cvu}
Volume 21 Issue2
May 2009
ISSN 1354-3164
www.accu.org

A Passion For It

It seems like a long time ago that I was first introduced to ACCU. Well, OK, it *was* a long time ago. Anyway, it was my first gig as a programmer, with memories of university still quite fresh in my mind, and I thought I was pretty good at it. I'd joined a small team of developers (I'm sure they all know who they are) and it didn't take them long to show me exactly how little I really knew. Fortunately they were, for the most part, quite gentle about it, and I, for my part, knew that I could be 'better' than I was. Even more fortunately, they all knew quite a lot – including how to join the ACCU, and get help from the mailing lists, the journals, and eventually, the conference.

My first conference was a very motivating experience, where I was lucky enough to meet some people who all shared a common goal – to be 'better'. Even more than that, it seemed that everyone who attended had a passion for it, that it was why they were there. It's not just about the formal sessions, either, as illuminating and interesting as they are. For me the most important aspect of the conference is the opportunity to meet other people involved with software development (not just developers) and, well, confer with them.

Now I still feel that I have a lot to learn – each conference I attend, each ACCU magazine article I read, every thread on the ACCU mailing lists reaffirms this for me – but I definitely feel that I am Better for being a member, and for participating in the community that is ACCU. Some of you will have attended the recent ACCU conference in Oxford. I hope that you, like me, feel 'better' for it.

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

ACCU exists for programmers at all levels of experience, from students and trainees to experienced developers. As well as publishing magazines, we run a respected annual developers' conference, and provide targeted mentored developer projects.

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

To find out more about ACCU's activities, or to join the organisation and subscribe to this magazine, go to www.accu.org.

Membership costs are very low as this is a non-profit organisation.

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 21.3: 1st June 2009
C Vu 21.4: 1st August 2009

Overload 92: 1st July 2009
Overload 93: 1st September 2009

WRITE FOR C VU

Both C Vu and Overload rely on articles submitted by you, the readers. We need articles at all levels of software development experience. What are you working on right now? Let us know!

Send articles to cvu@accu.org. The friendly magazine production team is on hand if you need help or have any queries.

ADVERTISE WITH US

COPYRIGHTS AND TRADE MARKS

Java Inheritance vs Composition

Paul Grenyer investigates some design issues.

In my article 'Boiler Plating Database Resource Clean Up Part II' [1], I discuss a way of writing JDBC boiler plate code using Execute Around Method [2]. Part of the suggested solution is a customisable error handling policy.

```
public interface ErrorPolicy
{
  void handleError(Exception ex);
  void handleCleanupError(Exception ex);
}
```

The implementer of the **ErrorPolicy** interface is free to handle errors in any way they choose (e.g. throw something, log something, ignore them, etc.). A distinction is made between an error caused by using or creating a JDBC object (**Connection**, **Statement**, **ResultSet**, etc.) and errors caused by releasing or cleaning up a JDBC object. Classes that use or create a JDBC object are required to implement the **ErrorPolicyUser** interface, so that they can be passed the error policy:

```
public interface ErrorPolicyUser
{
  void setErrorPolicy(ErrorPolicy errorPolicy);
}
```

Implementers of the **ErrorPolicyUser** interface are also free to do so in anyway they wish.

The example boiler plate I designed has a number of providers. The **ConnectionProvider**, for example, creates a JDBC **Connection** object, passes it to a **ConnectionUser** and then cleans up the object. The example in Listing 1 shows its usage.

As you can see, implementers of **ConnectionUser** must implement both the **use** method, which uses the JDBC **Connection** and the **setErrorPolicy** method as it extends **ErrorPolicyUser**. The example in Listing 2 shows one possible implementation of **ConnectionUser**.

This implementation has a single job to do, which is to execute an sql query using the supplied **Connection** object. Resource clean up in Java is a strange beast. Not only must it be carried out explicitly using **finally** (no **IDisposable**/**using** or destructors here!) the action of actually closing a resource can throw too. Therefore all sorts of error handling has to be implemented.

The example in Listing 2 is slightly contrived as my boiler plate will also take care of creating, executing and cleaning-up JDBC **Statement** objects, however I wanted to demonstrate exactly how the error policy fits in. As you can see, if an exception is thrown when creating or using the **Statement** object, the exception is passed to the **handleError** method of the exception policy. If there is an exception thrown on clean-up it is passed to the **handleCleanupError** method. A **DefaultErrorPolicy** is used by default, but the **setErrorPolicy** method allows a custom error policy to be set. The **setErrorPolicy** method's sole function is to receive and store the custom error policy. The reference that holds the error policy is also a member of the class.

The **setErrorPolicy** implementation and the **ErrorPolicy** reference must be repeated in every class that implements **ErrorPolicyUser**. That in itself isn't a huge amount of code, but what I object to is the unnecessary repetition.

Listing 1

```
final ConnectionProvider cp =
// Create a ConnectionProvider
   cp.provideTo(new ConnectionUser()
{
  @Override
  public void use(Connection con)
  {
    // use connection. }
  @Override
  public void setErrorPolicy(
     ErrorPolicy errorPolicy)
  {
    // store reference to error policy.
  }
});
```

Listing 2

```
public class User implements ConnectionUser
{
  private ErrorPolicy errorPolicy =
    new DefaultErrorPolicy();
  @Override
  public void use(Connection con)
  {
    try
    {
      final Statement stmt =
         con.createStatement();
      try
      {
        stmt.execute(sql);
      }
      finally
      {
        try
        {
          stmt.close();
        }
        catch(SQLException ex)
        {
          errorPolicy.handleCleanupError(ex);
        }
      }
    }
    catch(SQLException ex)
    {
      errorPolicy.handleError(ex);
    }
  }
  @Override
  public void setErrorPolicy(
     ErrorPolicy errorPolicy)
  {
    this.errorPolicy = errorPolicy;
  }
}
```

PAUL GRENYER

An active ACCU member since 2000, Paul is the founder of the Mentored Developers. Having worked in industries as diverse as direct mail, mobile phones and finance, Paul now works for a small company in Norwich writing Java. He can be contacted at paul.grenyer@gmail.com

Listing 3

```
public class ErrorPolicyUserImpl implements
   ErrorPolicyUser
{
  private ErrorPolicy errorPolicy;
  public ErrorPolicy getErrorPolicy()
  {
    return errorPolicy;
  }
  @Override
  public void setErrorPolicy(
     ErrorPolicy errorPolicy)
  {
    this.errorPolicy = errorPolicy;
  }
}
public class User implements ConnectionUser
{
  private ErrorPolicyUserImpl errorPolicyUser
     = new ErrorPolicyUserImpl();
  @Override
  public void use(Connection con)
  {
    try
    {
      final Statement stmt =
         con.createStatement();
      try
      {
        stmt.execute(sql);
      }
      finally
      {
        try
        {
          stmt.close();
        }
        catch(SQLException ex)
        {
          errorPolicyUser.getErrorPolicy()
             .handleCleanupError(ex);
        }
      }
    }
    catch(SQLException ex)
    {
      errorPolicyUser.getErrorPolicy()
         .handleError(ex);
    }
  }
  @Override
  public void setErrorPolicy(
     ErrorPolicy errorPolicy)
  {
    errorPolicyUser.setErrorPolicy(errorPolicy);
  }
}
```

The 'modern' Java way (I say 'modern' as I don't think it has always been this way) is to use composition in favour of inheritance. Java aside, this is generally good practice. A composition based solution would look something like Listing 3.

The composition solution is even more code in each **`ConnectionUser`** implementation and you still have a member and a method whose implementation must be repeated each time. The only advantage is that if you change the default error policy to a different type you only have to change it in one place. As, in most situations, the error policy will be set via **`setErrorPolicy`** anyway, this is no advantage at all.

So what about an inheritance based solution? Well, that would look something like Listing 4.

Listing 4

```
public abstract class AbstractErrorPolicyUser
   implements ErrorPolicyUser
{
  private ErrorPolicy errorPolicy =
     new DefaultErrorPolicy();
  protected ErrorPolicy getErrorPolicy()
  {
    return errorPolicy;
  }
  @Override
  public void setErrorPolicy(
     ErrorPolicy errorPolicy)
  {
    this.errorPolicy = errorPolicy;
  }
}
public class User extends AbstractErrorPolicyUser
   implements ConnectionUser
{
  @Override
  public void use(Connection con)
  {
    try
    {
      final Statement stmt
         = con.createStatement();
      try
      {
        stmt.execute(sql);
      }
      finally
      {
        try
        {
          stmt.close();
        }
        catch(SQLException ex)
        {
          getErrorPolicy()
             .handleCleanupError(ex);
        }
      }
    }
    catch(SQLException ex)
    {
      getErrorPolicy().handleError(ex);
    }
  }
}
```

With the inheritance solution the only repeated code is **`extends AbstractErrorPolicyUser`**. Everything else comes for free. Each **`ConnectionUser`** implementation has less code and significantly less repetition.

Java Inheritance

As stated in the main text, I see the lack of multiple inheritance in Java as a drawback. However, this is not a view held by everyone as Steve Love pointed out to me:

> 'I don't necessarily see it as a drawback as such – although it can be a limitation. Single inheritance has implications for this kind of policy inheritance (it's not really *is a*, more a *kind of* Mixin) but the argument goes that MI is so prone to error that causing extra typing is worth it.'

This is of course very true, but like any language feature, multiple inheritance is only dangerous if people don't know how to use it correctly.

A Case for Code Reuse

Pete Goodliffe shows us a useful case of the mythical 'code reuse'.

We hear a lot about a mythical thing called 'Code Reuse'. I'm not sold on it.

Re-use case 1: Code copied out of one app is surgically placed into another. Well, in my book that's less *code reuse* and more like *code re-purposing*. Or, less politely: *cut-and-paste programming*. It's often evil; tantamount to Code Piracy. Imagine a bunch of swashbuckling programmers pillaging and hoarding software gems from rich codebases around the seven software seas. Daring. But dangerous. It's coding with the bad hygiene of a salty seaman.

This kind of 'reuse' is a *real* killer when you've duplicated the same code fragment 516 times in one project and then discover it's got a bug. Having said that, cut-and-paste between projects does get stuff done. There's a lot of it about and the world hasn't come to a crashing end. Yet.

Cut-and-paste is a nasty business and no self-respecting programmer will admit to this kind of code reuse.

Re-use case 2: Consider a code library designed for inclusion in multiple projects. That's neater than cut-and-paste programming. That's theologically sound programming. But that's not code reuse. It's code *use*. The library was designed to be used like this from the very start.

See, I'm not sold on the whole code 'reuse' idea.

But I present here one genuine case for code reuse. I happen to like **Re-use case 3**. It's the best way to reuse code... Face it, almost all old code is a pile of pants; your old code is probably not good enough to be put into another application anyway, so don't try. (It's a miracle it worked anyway.)

However, you *can* reuse your old code: to learn how to build better code.

Re-use case 3: learning

We don't tend to look back over our own old code that often. We'd rather not think of the functional gremlins and typographical demons that lurk in our ancient handiwork. You thought it was perfect when you wrote it – but cast a critical eye over your old code and you'll inevitably bring to light all manner of code gotchas.

Programmers, as a breed, strive to move onwards: to learn new and exciting techniques, to face fresh challenges, and to solve more interesting problems. It's natural. Considering the rapid turnover in the job market, and the average duration of programming contracts, it's hardly surprising that very few software developers stick with the same codebase for a prolonged period of time.

But what does this do to the code we produce? What kind of *attitude* does it foster in our work? I maintain that exceptional programmers are

PETE GOODLIFFE

Pete Goodliffe is a software developer, columnist, speaker, and author who never stays at the same place in the software food chain. He has a passion for curry and doesn't wear shoes.. Pete can be contacted at pete@goodliffe.net

Java Inheritance vs Composition (continued)

The inheritance solution isn't without its disadvantages though.

If you want to implement `ConnectionUser` as an anonymous class, there is no way in Java to extend `AbstractErrorPolicyUser` and implement `ConnectionUser` at the same time. One simple solution is to have an intermediate class. (See Listing 5.)

Another drawback of Java is the lack of multiple inheritance (see sidebar). So if an implementation of `ConnectionUser` was required to extend another class, that would prevent it from also extending `AbstractErrorPolicyUser`. Under these circumstances composition should be used.

Listing 5

```
public abstract class AbstractConnectionUser
   extends AbstractErrorPolicyUser
   implements ConnectionUser
{}
new AbstractConnectionUser()
{
  @Override
  public void use(Connection con)
  {
    // ..
  }
};
```

it is wrong to follow 'prefer composition to inheritance' blindly

It is my opinion that, as with many blanket rules, it is wrong to follow 'prefer composition to inheritance' blindly and instead it should always be considered judiciously. I believe that the examples above demonstrate this. However, it cannot be ignored that if composition is generally preferred to inheritance the coupling implicit with inheritance is reduced and the single extended class permitted by Java is kept in reserve for when it is really needed. ■

Acknowledgements

Thank you to Richard Dehnen, Thomas Hawton, Seweryn Habdank-Wojewodzki and Steve Love for some very interesting and informative discussion that inspired this article.

References

[1] 'Boiler Plating Database Resource Clean Up Part 2' by Paul Grenyer http://www.marauder-consulting.co.uk/Boiler_Plating_Database_Resource_Cleanup_-_Part_II.pdf

[2] 'Execute Around Method – Another Tail of Two Patterns' by Kevlin Henney: http://www.two-sdg.demon.co.uk/curbralan/papers/AnotherTaleOfTwoPatterns.pdf

determined more by their attitude to the code they write and the way they write it, than by the actual code itself.

So the average programmer tends not to maintain their *own* code for too long. Rather than roll around in our own filth, we move on to new pastures and roll around in *someone else*'s filth. Nice. Of course, it's fun to complain about other people's poor code, but we easily forget how bad our own work was.

But of course, you'd never intentionally write bad code, would you?

Revisiting your old code can be an enlightening experience. It's like visiting an old relative you don't see very often. You don't know them as well as you think. You've forgotten things about them, about their funny quirks and irritating ways. And you're surprised at how they've changed since you last saw them (perhaps, for the worst).

Such evolution can unfortunately render your code anachronistic

Looking back at old code you've produced, you might shudder for a number of reasons...

Presentation

This is only really an issue for those languages which sanction ASCII-based artistic interpretation. Indeed, I rather admire Java and C# (for example) for having a de-facto standard presentation style. It avoids many of the fractures over coding styles found predominantly in the C and C++ camps.

For example, some C++ programmers follow standard library layout:

```
class standard_style
{
  int variable_name;
  bool method_name();
};
```

and some have more Java-esque leanings:

```
class JavaStyle
{
  int variableName;
  bool methodName();
};
```

I know over the years that my presentation style has changed wildly, depending on the company I'm working for at the time.

As long as the style is employed consistently in your codebase, this is a really trivial concern and nothing to be embarrassed about.

Listing 1

```
using System;
using System.Collections;

ArrayList list = new ArrayList(); // untyped
list.Add("Foo");
list.Add(Int(3)); // oops!
```

Listing 2

```
using System;
using System.Collections.Generic;

public class ModernLists
{
  static void Main()
  {
    List<string> list = new List<string>();
    list.Add("Foo");
    list.Add("Bar");
  }
}
```

The state of the art

Most languages have rapidly developing built-in libraries. Over the last 13 years the Java libraries have grown from a few hundred helpful classes to a veritable plethora of functionality. C# is a relative newcomer to the development world, but over three major revisions its library has burgeoned, and has grown a myriad of new facilities (including such excitement as generics, anonymous methods, iterators, partial types, and more).

Such evolution (which is especially rapid early in a language's development) can unfortunately render your code anachronistic. Anyone reading your code for the first time might presume that you didn't understand the new language/library features, when they simply did not actually exist when the code was written.

For example, C#'s generics means that the code you'd have written in 2004 in Listing 1 would today be written as in Listing 2. There is a very similar Java example with surprisingly similar class names! The state of the art moves much faster than your code.

I had a similar case recently when returning to a C++ codebase. Some years into its development, the Boost [1] libraries had been brought in. Several classes throughout the codebase had been tracking values that could have been out-of-date, so they had two variables, a boolean for validity, and a stored value (see Listing 3). Done once, it's a bit manual. Done all over the codebase it's a brittle structure that isn't immediately obvious, and easy to forget to check **`value`**'s validity before using it. However, once Boost had been brought in, all this was made easier with **`boost::optional`**,

Listing 3

```
class ValueCache
{
  bool value_valid;
  int  value;

  void ValueChanged(int new_value)
  {
    value_valid = true;
    value       = new_value;
  }

void UseValue()
  {
    DoSomething(value);
    // oops! we forgot to check if it was valid
    // first!
  }
};
```

Listing 4

```
class ValueCacheWithBoost
{
  boost::optional<int> value;
  void SetValue(int new_value)
  {
    value = new_value;
  }
  void UseValue()
  {
    DoSomething(value); // this does not
    // compile...
    // ... so you have to think about
    // whether value is valid
    if (value) DoSomething(*value)
  }
};
```

Listing 5

```
#define max(a,b) ((a)>(b)) ? (a) : (b)

void example()
{
  int a = 3, b = 10;
  int c = max(a, b);
}
```

Listing 6

```
template <typename T>
inline max(const T &a, const T &b)
{
  // Look mum! No brackets needed!
    return a > b ? a : b;
}

void better_example()
{
  int a = 3, b = 10;

  // this would have failed using the macro
  // because ++a would be evaluated twice
  int c = max(++a, b);
}
```

which says exactly what it's doing, and makes the code both safer, and more obvious (see Listing 4).

Idioms

It's perhaps most embarrassing to look back at old code, and see how un-idiomatic it is. If you now know more of the correct idiom for the language you're working with then old code can look quite, quite wrong.

Some time ago, I worked with a team of C programmers moving (shuffling slowly) towards the then brave new world of C++. One of their initial additions to the new codebase was a **`max`** helper, as in Listing 5. (Do you know why we have the brackets?)

After some time, someone revisited that early code, and knowing more about C++, realised how bad it actually was, and re-wrote it in more idiomatic C++, as in Listing 6. This actually fixed some very subtle lurking bugs, as shown in the listing.

Old code not looking how you'd write it now is actually a good thing

The original Listing 5 version also had another problem: the macro clobbers a name in the C++ standard library, which leads onto the even better solution in Listing 7: just use the **`std::max`** function that always existed. It's obvious in hindsight. That's the kind of thing you'd cringe about now, but had no idea about back in the day.

Design decisions

Did I really write that in Perl? Did I really use such a simplistic sorting algorithm? Did I really write that by hand, rather than just using a built-in library function?

Bugs

Perhaps this is the reason that drags you back to an old codebase. Sometimes coming back with fresh eyes uncovers obvious problems that you missed at the time.

Conclusion

Looking back over your old code is the best form of 'code reuse' I can think of. It's like a code review for yourself. It's a valuable exercise to do; perhaps you should take a quick tour through some of your old work. Do you like the way you used to program? Does your old code shape up in the modern world? (If it doesn't look too different from today's freshly minted lines of source, does that mean that you haven't learnt anything new recently?)

Listing 7

```
// don't declare any max function

void even_better_example()
{
  int a = 3, b = 10;
  int c = std::max(a,b);
}
```

So does this kind of thing matter? If your old code's not perfect, should you do anything about it? Should you go back and 'fix' the code? Probably not – if it ain't broke don't fix it. Sometimes the code does not rot, unless the world changes around it (compiler versions break your old code, or the latest library version no longer lets you compile).

It's important to understand how times have changed, how the programming world has moved on, and how your personal skills have improved over time. Old code not looking how you'd write it now is actually a good thing: it shows that you have learnt and improved from where you were. Perhaps you don't have the opportunity to revise it now, but knowing where you've come from helps to shape where you're going in your coding career. ■

Notes

[1] It's an excellent library, I strongly suggest you take a look if you're a C++ programmer.

Inside a Distributed Version Control System

Jim Hague shows us what goes on.

Grinton Lodge is a Youth Hostel that sits on an exposed hillside just above the small hamlet of Grinton in Swaledale, in the Yorkshire Dales National Park. A former Victorian shooting lodge, it now welcomes walkers and other travellers from around the world.

Tonight, a Wednesday in mid-November, is not one of its busiest nights. Kat, the duty staff member, tells me that there is a small corporate team-building group in the annex. There's no sign of them at present. Otherwise, that portion of the world that has beaten a path to the door of this grand building today consists of just me. And Kat goes home soon.

The November CVu, removed from its wrappers and read yesterday, lies in my bag. Taunting me. Go on, it says, if you're ever going to put finger to keyboard in the name of CVu, well, tonight you are out of excuses.

Bugger.

Let's look into Mercurial

If you're at all interested in version control systems – and any software developer not using one daily is a strange beast indeed – you'll at least have become vaguely aware in the last few years of the growing maturity of the latest group of version control systems offering funky new stuff. These are the distributed version control systems (DVCS). There is more to them than just their headline attributes, being able to check history and do checkins while disconnected from a central server, but these are damm useful to start with.

When I first heard about DVCS, it wasn't immediately obvious to me (to put it mildly) how they would work. After years of using a centralised version control system, I had a rough mental model of what went on. But how do you cope without the central server forcing ordering onto the changes?

Since then I've started using Mercurial (http://www.selenic.com/mercurial). Mercurial is a DVCS. It's one of three DVCSs that have gained significant popularity in the last few years, the other two being Git (http://git-scm.com) and Bazaar (http://bazaar-vcs.org/). I switched a significant work project over to Mercurial (from Subversion) in mid-1997, because a customer site required on-site work but could not allow access back to the company VPN. I chose Mercurial for a variety of reasons, which I won't bore you with here. If you must know, see the box.

What I want to do in this article is give you an insight into how a DVCS works. OK, so specifically I'm going to be talking about Mercurial, but Git and Bazaar attack the problem in a similar way. But first I'd better give you some idea of how you use Mercurial.

The 5 minute Mercurial overview

I think it unlikely that someone possessing the taste and discernment to be reading CVu would not be familiar with at least one version control system. So, while I want to give you a flavour of what it's like to use, I'm not going to hang about. If you'd like a proper introduction, or you don't follow something, I thoroughly recommend you consult the Mercurial book.

To start using Mercurial to keep track of a project:

JIM HAGUE

Jim has coded (and the rest) for companies large and small since the mid-80's, but only discovered ACCU in 2002. Five years ago he somehow fell into working on ATC systems and is still there. Email him at jim.hague@acm.org

OK, if you must know...

- **Implementability**. I needed the system to work on Windows, Linux and AIX. The latter was not one of the directly supported platforms for any of the candidates. Git's implementation uses a horde of tools. Bazaar requires only Python, but required Python 2.4 while IBM stubbornly still supplies only Python 2.3. Mercurial requires Python 2.3 or greater, and uses some C for speed.
- **Simplicity**. My users used Subversion daily, but did not generally have much experience with other VCS. From the command line, Mercurial's core operations will be familiar to a Subversion user. This is also true of Bazaar, but was less true of Git. Git has improved in this matter since then, but a Dr Winder of this parish tells me that it's still possible to seriously embarrass yourself. There was also a lack of Windows support for Git at the time.
- **Speed**. Mercurial is fast. In the same ballpark as Git. Bazaar wasn't, and although it has improved significantly, has, in my estimation, added user complexity in the process, and at the time of writing is still off the pace for some operations.
- **Documentation**. At the time, Bryan O'Sullivan's excellent Mercurial book (http://hgbook.red-bean.com) was a clear winner for best documentation.

```
$ hg init
$
```

This creates the repository root in the current directory.

Like CVS (http://www.nongnu.org/cvs/) with its `CVS` directory and Subversion (http://subversion.tigris.org/) with its `.svn` directory, Mercurial keeps its private data in a directory. Mercifully there is only one of these, in the top level of your project. And rather than holding details of where the actual repository is to be found, the `.hg` directory holds the entire repository.

Next you need to specify the files you want Mercurial to track.

```
$ echo "There was a gibbon one morning"
   > pome.txt
$ hg add pome.txt
$
```

As you might expect, this marks the files as to be added. And as you might also expect, you need to commit to record the added files in the repository. The commit comment can be supplied on the command line; if you don't supply a comment, you'll be dropped into an editor to provide one.

There is a suggested format for these messages – a one line summary followed by any more required detail on following lines. By default Mercurial will only display the first line of commit messages when listing changes. In these examples I'll stick to terse messages, and I'll enter them from the command line.

```
$ hg commit -m "My Pome" -u "Jim Hague
<jim.hague@acm.org>"
$
```

Mercurial records the user making the change as part of the change information. It is usual to give your name and email address as I've done here. You can imagine, though, that constantly having to repeat this is a bit tedious, so you can set a default user name in a configuration file. Mercurial keeps global, user and repository configurations, and it can go in any of those.

As with Subversion, after further edits you see how your working copy differs from the repository (Listing 1), and look through a log of changes (Listing 2). There are some items here that need an explanation.

Listing 1

```
$ hg status
M pome.txt
$ hg diff
diff -r 33596ef855c1 pome.txt
--- a/pome.txt  Wed Apr 23 22:36:33 2008 +0100
+++ b/pome.txt  Wed Apr 23 22:48:01 2008 +0100
@@ -1,1 +1,2 @@ There was a gibbon one morning
There was a gibbon one morning
+said "I think I will fly to the moon".
$ hg commit -m "A great second line"
$
```

Listing 2

```
$ hg log
changeset:   1:3d65e7a57890
tag:         tip
user:        Jim Hague <jim.hague@acm.org>
date:        Wed Apr 23 22:49:10 2008 +0100
summary:     A great second line

changeset:   0:33596ef855c1
user:        Jim Hague <jim.hague@acm.org>
date:        Wed Apr 23 22:36:33 2008 +0100
summary:     My Pome

$
```

The **changeset** identifer is in fact two identifiers separated by a colon. The first is the sequence number of the changeset in the repository, and is directly comparable to the change number in a Subversion repository. The second is a globally unique identifier for that change. As the change is copied from one repository to another (this is a distributed system, remember, even if we haven't come to that bit yet), its sequence number in any particular repository will change, but the global identifier will always remain the same. **tip** is a Mercurial term. It means simply the most recent change.

Want to rename a file?

```
$ hg mv pome.txt poem.txt
$ hg status
A poem.txt
R pome.txt
$ hg commit -m "Rename my file"
$
```

(The command to rename a file is actually **hg rename**, but Mercurial saves Unix-trained fingers from typing embarrassment.)

At this point you may be wondering about directories. **hg mkdir** perhaps? Well, no. Mercurial only tracks files. To be sure, the directory a file occupies is tracked, but effectively only as a component of the file name. This has the slightly unexpected result that you can't record an empty directory in your repository. (I tripped over this converting a work Subversion repository. One possibility is to create a placeholder file in the directory. In the event I created the directory (which receives build products) as part of the build instead.)

Given this, and the status output above that suggests strongly that Mercurial treats a rename as a copy followed by a delete, you may be worried that Mercurial won't cope at all well with rearranging your repository. Relax. Mercurial does store the details of the rename as part of the changeset, and copes very well with rearrangements. (The Mercurial designers justify not dealing with directories as first class objects by pointing out that provided you can correctly move files about in the tree, the other reasons for tracking directories are uncommon and do not in their opinion justify the considerable added complexity. So far I've found no reason to doubt that judgement.)

Want to rewind the working copy to a previous revision? **hg update** (Listing 3) updates the working files. In this case I'm specifying that I want

Listing 3

```
$ hg update -r 1
1 files updated, 0 files merged, 1 files removed,
0 files unresolved
$
```

Listing 4

```
elsewhere$ cd Jim-Poem
elsewhere$  hg log
changeset:   3:a065eb26e6b9
tag:         tip
user:        Jim Hague <jim.hague@acm.org>
date:        Thu Apr 24 18:52:31 2008 +0100
summary:     Rename my file

changeset:   2:ff97668b7422
user:        Jim Hague <jim.hague@acm.org>
date:        Thu Apr 24 18:50:22 2008 +0100
summary:     Finished first verse

changeset:   1:3d65e7a57890
user:        Jim Hague <jim.hague@acm.org>
date:        Wed Apr 23 22:49:10 2008 +0100
summary:     A great second line

changeset:   0:33596ef855c1
user:        Jim Hague <jim.hague@acm.org>
date:        Wed Apr 23 22:36:33 2008 +0100
summary:     My Pome
```

to go back to local changeset 1. I could also have typed **-r 3d65e7a57890**, or even **-r 3d**; when specifying the global change identifier you only need to type enough digits to make it unique.

This is all very well, but it's not exactly distributed, is it?

Copy an existing repository:

```
elsewhere$ hg clone ssh://jim.home.net/Poem Jim-
Poem
updating working directory
1 files updated, 0 files merged, 0 files removed,
0 files unresolved
```

(You can access other repositories via the file system, over http or over ssh – see Listing 4.)

hg clone is aptly named. It creates a new repository that contains exactly the same changes as the source repository. You can make a clone just by copying your project directory, if you're confident nothing else will access it during the copy. **hg clone** saves you this worry, and sets the default push/pull location in the new repo to the cloned repo.

From that point, you use **hg pull** to collect changes from other places into your repo (though note it does not by default update your working copy), and, as you might guess, **hg push** shoves your changes into a foreign repository. By default these will act on the repository you cloned from, but you can specify any other repository.

More on those in a moment. First, though, I want to show you something you can't do in Subversion. Start with the repository with 4 changes we just cloned. I want to focus on the first couple of lines, so I'll wind the working copy back to the point where only those lines exist:

```
$ hg update -r 1
1 files updated, 0 files merged, 1 files removed,
0 files unresolved
$
```

and make a change (Listing 5).

The alert among you will have sat up at that. Well done! Yes, there's something very worrying. How can I commit a change at an old point?

Listing 5

```
$ hg diff
diff -r 3d65e7a57890 pome.txt
--- a/pome.txt  Wed Apr 23 22:49:10 2008 +0100
+++ b/pome.txt  Thu Apr 24 19:13:14 2008 +0100
@@ -1,2 +1,2 @@ There was a gibbon one morning
-There was a gibbon one morning
-said "I think I will fly to the moon".
+There was a baboon who one afternoon
+said "I think I will fly to the sun".
$ hg commit -m "Better first two lines"
$
```

Listing 6

```
$ hg heads
changeset:   4:267d32f158b3
tag:         tip
parent:      1:3d65e7a57890
user:        Jim Hague <jim.hague@acm.org>
date:        Thu Apr 24 19:13:59 2008 +0100
summary:     Better first two lines

changeset:   3:a065eb26e6b9
user:        Jim Hague <jim.hague@acm.org>
date:        Thu Apr 24 18:52:31 2008 +0100
summary:     Rename my file

$
```

If you try this in Subversion, it will complain mightily about your file being out of date. But Mercurial just went ahead and did something. The Bazaar experts among you will know that in Bazaar, if you use **`bzr revert -r`** to bring the working copy to a past revision, make a change and commit, then your latest version will be the past revision plus your change. Perhaps that's what Mercurial did?

No. What Mercurial did is central to Mercurial's view of the world. You took your working copy back to an old changeset, and then committed a fresh change based at that changeset. Mercurial actually did just what you asked it to do, no more and no less. Let's see the initial evidence (Listing 6).

Listing 7

```
$ hg glog
@  changeset:   4:267d32f158b3
|  tag:         tip
|  parent:      1:3d65e7a57890
|  user:        Jim Hague <jim.hague@acm.org>
|  date:        Thu Apr 24 19:13:59 2008 +0100
|  summary:     Better first two lines
|
| o  changeset:   3:a065eb26e6b9
| |  user:        Jim Hague <jim.hague@acm.org>
| |  date:        Thu Apr 24 18:52:31 2008 +0100
| |  summary:     Rename my file
| |
| o  changeset:   2:ff97668b7422
|/   user:        Jim Hague <jim.hague@acm.org>
|    date:        Thu Apr 24 18:50:22 2008 +0100
|    summary:     Finished first verse
|
o  changeset:   1:3d65e7a57890
|  user:        Jim Hague <jim.hague@acm.org>
|  date:        Wed Apr 23 22:49:10 2008 +0100
|  summary:     A great second line
|
o  changeset:   0:33596ef855c1
   user:        Jim Hague <jim.hague@acm.org>
   date:        Wed Apr 23 22:36:33 2008 +0100
   summary:     My Pome

$
```

Listing 8

```
$ hg merge
merging pome.txt and poem.txt
0 files updated, 1 files merged, 0 files removed,
0 files unresolved
(branch merge, don't forget to commit)
$ cat poem.txt
There was a baboon who one afternoon
said "I think I will fly to the sun".
So with two great palms strapped to his arms,
he started his takeoff run.
$ hg commit -m "Merge first line branch"
$
```

Time for some more Mercurial terminology. You can think of a **`head`** in Mercurial as the most recent change on a branch. In Mercurial, a branch is simply what happens when you commit a change that has as its parent a change that already has a child. Mercurial has a standard extension **`hg glog`**, which uses some ASCII art to show the current state (Listing 7).

`hg view` shows a nicer graphical view. (Though, being Tcl/Tk based, not that much nicer.)

So the change is in there. It's the latest change, and is simply on a different branch to the other changes.

Almost invariably, you will want to bring everything back together and merge the branches. A merge is a change that combines two heads back into one. It prepares an updated working directory with the merged contents of the two heads for you to review and, if satisfactory, commit. (Listing 8.)

Listing 9

```
$ hg glog
@    changeset:   5:792ab970fc80
|\   tag:         tip
| |  parent:      4:267d32f158b3
| |  parent:      3:a065eb26e6b9
| |  user:        Jim Hague <jim.hague@acm.org>
| |  date:        Thu Apr 24 19:29:53 2008 +0100
| |  summary:     Merge first line branch
| |
| o  changeset:   4:267d32f158b3
| |  parent:      1:3d65e7a57890
| |  user:        Jim Hague <jim.hague@acm.org>
| |  date:        Thu Apr 24 19:13:59 2008 +0100
| |  summary:     Better first two lines
| |
o |  changeset:   3:a065eb26e6b9
| |  user:        Jim Hague <jim.hague@acm.org>
| |  date:        Thu Apr 24 18:52:31 2008 +0100
| |  summary:     Rename my file
| |
o |  changeset:   2:ff97668b7422
|/   user:        Jim Hague <jim.hague@acm.org>
|    date:        Thu Apr 24 18:50:22 2008 +0100
|    summary:     Finished first verse
|
o  changeset:   1:3d65e7a57890
|  user:        Jim Hague <jim.hague@acm.org>
|  date:        Wed Apr 23 22:49:10 2008 +0100
|  summary:     A great second line
|
o  changeset:   0:33596ef855c1
   user:        Jim Hague <jim.hague@acm.org>
   date:        Wed Apr 23 22:36:33 2008 +0100
   summary:     My Pome

$
```

Listing 9 is the ASCII art again showing what just happened. Oh, and notice in the above that Mercurial has done the right thing with regard to the rename.

So, our little branch change has now been merged back, and we have a single line of development again. Notice that unlike the other changesets, changeset 5 has two parent changesets, indicating it is a merge changeset. You can only merge two branches in one operation; or putting it another way, a changeset can have a maximum of two parents.

This behaviour is absolutely central to Mercurial's philosophy. If a change is committed that takes as its starting point a change that already has a child, then a branch gets created. Working with Mercurial, branches get created frequently, and equally frequently merged back. As befits any frequent operation, both are easy to do.

You're probably thinking at this point that this making a commit onto an old version is a slightly strange thing to do, and you'd be right. But that's exactly what's going to happen the moment you go distributed. Two people working independently with their own repositories are going to make commits based, typically, on the latest changes they happen to have incorporated into their tree. To be Distributed, a DVCS has to deal with this. Mercurial faces it head-on. When you pull changes into your repo (or someone else pushes them), if any of the changes overlap – are both based on the same base change – you get extra heads, and it's up to you to let these extra heads live or merge, as you please.

In practice this is more manageable then you might think. Consider a typical Mercurial usage, where the 'master' repo sits on a known server, and everyone pulls changes from the master and pushes their own efforts to the master. By default Mercurial won't let you push if the receiving repo will gain an extra head as a result, so you typically pull (and do any required merging) just before pushing. Subversion users will recognise this pattern. Subversion won't let you commit a change if your working copy is not at the very latest revision, so the Subversion user will update, and merge if necessary, just before committing.

What, then, about a branch in the conventional sense of '1.0 maintenance branch'? Typically in Mercurial you'd handle this by keeping a separate cloned repository for those changes. Cloning is fast, and if local, uses hard links where possible on filesystems that support them, so isn't necessarily extravagant on disc space. You can, if you prefer, handle them all in a single repo with 'named branches', but cloning is definitely simpler.

OK, so now you know the basics of using Mercurial. We can proceed to looking at how this magic is achieved. In particular, where does this magic globally unique identifier for a change come from?

Inside the Mercurial repo

The way Mercurial handles its repo is really quite simple.

That's simple, as in 'most things are simple once you know the answer'. I found the explanation helpful, so this section attempts the 10,000ft (FL100 if you prefer) view of Mercurial. (For the curious, Bryan O'Sullivan's excellent Mercurial book has a chapter on the subject, and the Mercurial website has a fair amount of detail too.)

First remember that any file or component can only have one or two parents. You can't merge more than one other branch at once.

We start with the basic building block, which Mercurial calls a revlog. A revlog is a thing that holds a file and all the changes in the file history. (For any non-trivial file, this will actually be two files on the disc, a data file and an index). The revlog stores the differences between successive versions of the file, though it will periodically store a complete version of the file instead of a difference, so that the content of any particular file version can always be reconstructed without excessive effort.

Under the secret-squirrel Mercurial `.hg` directory at the top of your project is a store which holds a revlog for each file in your project. So you have the complete history of the project locally. No more round trips to the server.

Listing 9

```
$ hg debugindex .hg/store/data/pome.txt.i
   rev    offset  length   base linkrev nodeid       p1           p2
     0         0      32      0       0 6bbbd5d6cc53 000000000000 000000000000
     1        32      51      0       1 83d266583303 6bbbd5d6cc53 000000000000
     2        83      84      0       2 14a54ec34bb6 83d266583303 000000000000
     3       167      76      3       4 dc4df776b38b 83d266583303 000000000000
$
```

Both the differences between successive versions and the periodic complete versions of a file are compressed before storing. This is surprisingly effective at minimising the storage requirements of this entire history of your project. I have a small Java project handy, comprising a little over 300 source modules. There are 5 branches plus the mainline, and some 1920 commits in all. A Subversion checkout of the current mainline takes 51Mb. Converting the project to Mercurial yields a Mercurial repository that takes 60Mb, so a little bigger. Remember, though, that the Mercurial repository includes not just the working copy, but also the entire history of the project.

Any point in the evolution of a revlog can be uniquely identified with a nodeid. This is simply the SHA1 hash of the current file contents concatenated with the nodeids of one or both parents of the current revision. Note that this way, two file states are identical if and only if the file contents are the same ***and*** the file has the same history.

Listing 10 shows a dump of a revlog index.

Note here that a file state can have two parents. If both the parent nodeids are non-null, the file state has two parents, and the state is therefore the result of a merge.

Let's dump out a revlog at a particular revision:

```
$ hg debugdata .hg/store/data/pome.txt.i 2
There was a gibbon one morning
said "I think I will fly to the moon".
So with two great palms strapped to his arms,
he started his takeoff run.
$
```

The next component is the manifest. This is simply a list of all the files in the project, together with their current nodeids. The manifest is a file, held in a revlog. The nodeid of the manifest, therefore, identifies the project filesystem at a particular point.

```
$ hg debugdata .hg/store/00manifest.i 5
poem.txt5168b1a5e2f44aa4e0f164e170820845183f50c8
$
```

Finally we have the changeset. This is the atomic collection of changes to a repository that leads to a new revision. The changeset info includes the nodeid of the corresponding manifest, the timestamp and committer ID, a list of changed files and a comment. The changeset also includes the nodeid of the parent changeset, or the two parents if the change is a merge. The changeset description is held in a revlog, the changelog.

```
$ hg debugdata .hg/store/00changelog.i 5
1ccc11b6f7308cc8fa1573c2f3811a4710c91e3e
Jim Hague <jim.hague@acm.org>
1209061793 -3600
poem.txt
pome.txt

Merge first line branch
$
```

The nodeid of the changeset, therefore, gives us a globally unique identifier for any particular change. Changesets have a Subversion-like incrementing change number, but it is peculiar to that repository. The nodeid, however, is global.

One more detail remains to complete the picture. How do we get back from a particular file change to find the responsible changeset? Each revlog change has a linkrev entry that does just this.

So, now we have a repository with a history of the changes applied to that repository. Each change has a unique identifier. If we find that change in another repository, it means that at the point in the other repository we have exactly the same state; the file contents and history are identical.

At this point we can see how pulling changes from another repository works. Mercurial has to determine which changesets in the source repository are missing in the target repository. To do this, for each head in the source repo it has to find the most recent change in that head that is already present in the target repo, and get any remaining changes after that point. These changes are then copied over and applied.

The Mercurial revlog format has proved remarkably durable. Since the first release of Mercurial in April 2005, there have been a total of 5 changes to the file format. However, of those, all but one have been changes to the handling of file names. The most recent change, in October 2008, and its predecessor in December 2006, were both introduced purely to cope with Windows specific issues. The one change that touched the datastructures described above was in April 2006. The format introduced, RevLogNG, changed only the details of index data held, not the overall design. The chief Mercurial developer, Matt Mackall, notes that the code in present-day Mercurial devoted to reading the old format comprises 28 lines of Python. Compared with, say, the early tribulations of Subversion and the switch from bdfs to fsfs, this is an impressive record.

Reflections on going distributed

It's nearly traditional at this stage in an introduction to DVCS to demonstrate several different workflow scanarios that you can build with a DVCS. Which makes the important point that a DVCS can be adapted to your workflow in a way that is at best unwieldy with a CVCS. I intend, though, to break with tradition here.

By this stage, I hope you can see that distributing version control works by introducing branches where development takes place in parallel. Mercurial treats these branches as arising naturally from the commits made and transferred between repositories. Both Git and Bazaar take a slightly different viewpoint, and explicitly generate a fresh branch for work in a particular repository. But in both cases the underlying principle of identifying changes by a globally unique identifier and resolving parallel development by merges between overlapping changes is the same. And all three can be used in a truly distributed manner, with full history and the ability to commit being available locally.

So instead of chatter on about workflows, I want instead to reflect on the consequences all this has for that all-important question of whether a DVCS is a suitable vehicle for your data.

The first is a minor and rather obvious point. If you want to store files that are very large and which change often in your DVCS, then all the compression in the world is unlikely to stop the storage requirements for the full project history from becoming uncomfortably large, particularly if the files are not very compressible to start with.

The second, and main, point is that there is an important question you need to ask about your data. We've seen that a DVCS relies on branching and merging to weave its magic. So take a close look at your data, and ask: '*Will it merge?*'

The subset of plain old text which comprises program source code requires some human oversight, but will merge automatically well enough for the process to be well within the bounds of the possible.

Unfortunately when we move further afield mergeability becomes a rarer commodity. I nearly began the previous paragraph by stating that plain old text will merge well enough. Then Doubt set in – what about XML? Or BASE64 encoded content?

Of course, merge doesn't necessarily have to be textual merge. I am told that Word can be used to diff and merge two Word .doc files, a data format notorious for its binary impenetrability. As long as some suitable merge agent is available, and the DVCS can be configured to use it for data of a particular type (Mercurial can have the merge and diff tools specified with reference to the file extension on which they operate – I assume Bazaar and Git are similar), then there is no bar to successful DVCS use.

Before this reliance on mergeability causes you to dismiss DVCS out of hand, reflect. A CVCS can only handle non-mergeable data by acting as a versioned file store; in other words, having as the only available merge option the use of one or other of the merge candidates in its entireity. Useful though a versioned file store can be, it cannot be considered a full-featured version control system. By treating the offending unmergeable files as external to the DVCS, or with careful workflow – disabling the distributed and mergeable potentials – a DVCS can deal with these files, but only at a cost of its distributedness or its version control system-ness. In this it differs little from a CVCS.

So, for all data you want to version control, let your battle cry be: '*Will it merge?*'

At this point, I have an urge to don lab coat and safety goggles and be videoed attempting to mechanically merge data in a variety of different formats. Frankly, this is unlikely to be as exciting as blending iPhones (Ref: www.willitblend.com), but from a system development point of view it's rather more important. And, I think gives us a large clue as to one of the reasons for the continuing popularity of Plain Old Text as a source code representation mechanism. ■

Note

I'm no poet. The poem is, of course, *Silly Old Baboon* by the late, great, Spike Milligan. From *A Book of Milliganimals*, Puffin, 1971.

The Case for D

Andrei Alexandrescu presents the evidence for a new language.

Let's see why the D programming language is worth a serious look.

I'm not deluding myself that it's an easy task to convince you. We programmers are a strange bunch in the way we form and keep language preferences. The knee-jerk reaction of a programmer when eyeing a *The XYZ Programming Language* book on a bookstore shelf is something like, 'All right. I'll give myself 30 seconds to find something I don't like about XYZ.' Acquiring a programming language is a long and arduous process, and long-term satisfaction is delayed and uncertain. Trying to find quick reasons to avoid such an endeavor is a survival instinct: the stakes are high and the investment is risky, so having the ability to make a rapid negative decision early in the process can be a huge relief.

That being said, learning and using a programming language can be a lot of fun. By and large, coding in a language is fun if the language does a satisfactory job at fulfilling the principles that the coder using it holds in high esteem. Any misalignment would cause the programmer to regard the language as, for example, sloppy and insecure or self-righteous and tedious. A language can't possibly fulfill everyone's needs and taste at the same time as many of them are contradictory, so it must carefully commit to a few fundamental coordinates that put it on the landscape of programming languages.

So what's the deal with D? You might have heard of it already – the language with a name like a pun taken a bit too far; annoyingly mentioned now and then on newsgroups dedicated to other languages before the off topic police reprimand the guilty; praised by an enthusiastic friend all too often; or simply as the result of an idle online search à la 'I bet some loser on this big large Internet defined a language called D, let's see... oh, look!'

This article is a very broad overview, so by necessity it uses concepts and features without introducing them rigorously as long as they are reasonably intuitive. It also has no bibliographical references, but it does mention all the proper terms, and google.com can do wonders when asked the right questions.

Let's take a brief look at some of D's fundamental features. Be warned that many features or limitations come with qualifications that make their boundaries fuzzy. So if you read something that doesn't quite please you, don't let that bother you too much: the next sentence may contain a redeeming addendum. For example, say you read "D has garbage collection" and you get a familiar frozen chill up the spine that stops in the cerebrum with the imperious command 'touch the rabbit foot and stay away.' If you are patient, you'll find out that D has constructors and destructors with which you can implement deterministic lifetime of objects.

But before getting into it

Before getting into the thick of things, there are a few things you should know. First and foremost, if you kind of considered looking into D for whatever reason, this time is not 'as good as any,' it's in fact much better than others if you're looking for the edge given by early adoption. D has been evolving at a breakneck pace but in relative silence, and a lot of awesome things have been and are being done about it that are starting to become known just about now – some literally in this very article. At the time of this writing, the book *The D Programming Language* is one-third complete and due to appear in a couple of months.

This state of transition as of now is putting yours truly in the unenviable position of dealing with a moving target. I opted for writing an article that ages nicely at the expense of being occasionally frustrating in that it describes features that haven't been implemented yet or are incompletely implemented.

There are two major versions of the language, D1 and D2. This article focuses on D2 exclusively. D1 is stable (will undergo no other changes but bug fixes), and D2 is a major revision of the language that sacrificed some backwards compatibility for the sake of doing things consistently right, and for adding a few crucial features related to manycores and generic programming. In the process, the language's complexity has increased, which is in fact a good indicator because no language in actual use has ever gotten smaller. Even languages that started with the stated intent to be 'small and beautiful' inevitably grew with use. (Yes, even Lisp. Spare me.) Although programmers dream of the idea of small, simple languages, when they wake up they seem to only want more modeling power.

The official D compiler is available off digitalmars.com for free on major desktop platforms (Windows, Mac, and Linux). Other implementations are well underway, notably including a .NET port. There are also two essential D libraries, the official one called Phobos, and an industrial-strength library called Tango. Tango, designed for D1, is being ported to D2, and Phobos (which was frustratingly small and quirky in its D1 iteration) underwent major changes and additions to take full advantage of D2's capabilities. (There is, unsurprisingly, an amount of politics and bickering about which library is better, but competition seems to spur both into being as good as they can be.)

Last but definitely not least, the immensely popular Qt windowing library has recently released a D binding (in alpha as of this writing). This is no small news as Qt is a great (the best if you listen to the right people) library for developing GUI applications portable to pretty much every OS with a pulse. The D bindings for Qt fully take D into 'the GUIth dimension', which completes the language's offering quite spectacularly. Even better, the development came shortly after Qt relaxed its license by adding LGPL to the available licenses. This means that commercial applications can use Qt without restrictions and without paying a licensing fee.

D fundamentals

D could be best described as a high-level systems programming language. It encompasses features that are normally found in higher-level and even scripting languages – such as a rapid edit-run cycle, garbage collection, built-in hashtables, or a permission to omit many type declarations – but also low-level features such as pointers, storage management in a manual (à la C's `malloc`/`free`) or semi-automatic (using constructors, destructors, and a unique `scope` statement) manner, and generally the same direct relationship with memory that C and C++ programmers know and love. In fact, D can link and call C functions directly with no intervening translation layer. The entire C standard library is directly available to D programs. However, you'd very rarely feel compelled to go that low because D's own facilities are often more powerful, safer, and just

ANDREI ALEXANDRESCU

Andrei Alexandrescu has been in his last quarter of PhD candidacy at the University of Washington for the last three quarters. Finally figuring out that a dissertation is needed to be let go, he wrote one. So he will finish Real Soon Now, quite fit for the worst time ever to look for jobs. His book, *The D Programming Language*, is out this year.

as efficient. By and large, D makes a strong statement that convenience and efficiency are not necessarily at odds.

D is multi-paradigm, meaning that it fosters writing code in object-oriented, functional, generic, and procedural style within a seamless and remarkably small package. The following bite-sized sections give away some generalities about D.

Hello, cheapshot

Let's get that pesky syntax out of the way. So, without further ado:

```
import std.stdio;
void main()
{
  writeln("Hello, world!");
}
```

Syntax is like people's outfits – rationally, we understand it shouldn't make much of a difference and that it's shallow to care about it too much, but on the other hand we can't stop noticing it. (I remember the girl in red from *The Matrix* to this day.) For many of us, D has much of the 'next door' familiar looks in that it adopted the C-style syntax also present in C++, Java, and C#. (I assume you are familiar with one of these, so I don't need to explain that D has par for the course features such as integers, floating-point numbers, arrays, iteration, and recursion.)

Speaking of other languages, please allow a cheapshot at the C and C++ versions of "Hello, world." The classic C version, as lifted straight from the second edition of K&R, looks like this:

```
#include <stdio.h>
main()
{
  printf("hello, world\n");
}
```

and the equally classic C++ version is (note the added enthusiasm):

```
#include <iostream>
int main()
{
  std::cout << "Hello, world!\n";
}
```

Many comparisons of the popular first program written in various languages revolve around code length and amount of information needed to understand the sample. Let's take a different route by discussing

D systematically attempts to make the right thing synonymous to the path of least resistance

correctness, namely: what happens if, for whatever reason, writing the greeting to the standard output fails? Well, the C program ignores the error because it doesn't check the value returned by `printf`. To tell the truth, it's actually a bit worse; although on my system it compiles flag-free and runs, C's "hello world" returns an unpredictable number to the operating system because it falls through the end of `main`. (For me, it always returns 13 on Ubuntu, which got me a little scared.) It turns out that the program as written is not even correct under the C89 or C99 standards. After a bit of searching, The Internet seems to agree that the right way to open the hailing frequencies in C is:

```
#include <stdio.h>
int main()
{
  printf("hello, world\n");
  return 0;
}
```

Syntax is like people's outfits – it's shallow to care about it too much, but on the other hand we can't stop noticing it

which does little in the way of correctness because it replaces an unpredictable return with one that always claims success, whether or not printing succeeded.

The C++ program is guaranteed to return 0 from `main` if you forgot to return, but also ignores the error because, um, at program start `std::cout.exceptions()` is zero and nobody checks for `std::cout.bad()` after the output. So both programs will claim success even if they failed to print the message for whatever reason. The corrected C and C++ versions of the global greet lose a little of their lip gloss:

```
#include <stdio.h>
int main()
{
  return printf("hello, world\n") < 0;
}
```

and

```
#include <iostream>
int main()
{
  std::cout << "Hello, world!\n";
  return std::cout.bad();
}
```

Further investigation reveals that the classic "hello, world" for other languages such as Java (code omitted due to space limitations), J# (a language completely, I mean *completely* unrelated to Java), or Perl, also claim success in all cases. You'd almost think it's a conspiracy, but fortunately the likes of Python and C# come to the rescue by throwing an exception.

How does the D version fare? Well, it doesn't need any change: `writeln` throws on failure, and an exception issued by `main` causes the exception's message to be printed to the standard error stream (if possible) and the program to exit with a failure exit code. In short, the right thing is done automatically. I wouldn't have taken this cheapshot if it weren't for two reasons. One, it's fun to imagine the street riots of millions of betrayed programmers crying how their "Hello, world" program has been a sham. (Picture the slogans: "Hello, world! Exit code 13. Coincidence?" or "Hello, sheeple! Wake up!" etc.) Second, the example is not isolated, but illustrative for a pervasive pattern – D attempts not only to allow you to do the right thing, it systematically attempts to make the right thing synonymous to the path of least resistance whenever possible. And it turns out they can be synonymous more often than one might think. (And before you fix my code, `void main()` is legal D and does what you think it should. Language lawyers who destroy noobs writing `void main()` instead of `int main()` in C++ newsgroups would need to find another favorite pastime if they switch to D.)

Heck, we planned to discuss syntax and ended up with semantics. Getting back to syntax, there is one notable departure from C++, C#, and Java: D uses `T!(X, Y, Z)` instead of `T<X, Y, Z>` (and `T!(X)` or simply `T!X` for `T<X>`) to denote parameterized types, and for good reasons. The choice of angular brackets, when clashing with the use of `<`, `>`, and `>>` as arithmetic operands, has been a huge source of parsing problems for C++, leading to a hecatomb of special rules and arbitrary disambiguations, not to mention the world's least known syntax `object.template fun<arg>()`. If one of your C++ fellow coders has Superman-level confidence, ask them what that syntax does and you'll see Kryptonite at work. Java and C# also adopted the angular brackets but wisely chose to disallow arithmetic expressions as parameters, thus preemptively crippling the option to ever add them later. D extends the traditional unary

operator ! to binary uses and goes with the classic parentheses (which (I'm sure) you *always* pair properly).

Compilation model

D's unit of compilation, protection, and modularity is the file. The unit of packaging is a directory. And that's about as sophisticated as it goes. There's no pretense that the program source code would really feel better in a super-duper database. This approach uses a 'database' tuned by the best of us for a long time, integrating perfectly with version control, backup, OS-grade protection, journaling, what have you, and also makes for a low entry barrier for development as all you need is an editor and a compiler. Speaking of which, specialized tool support is at this time scarce, but you can find things like the emacs mode `d-mode`, the Eclipse plugin Descent, the Linux debugger ZeroBugs, and the full IDE Poseidon.

Generating code is a classic two-stroke compile and link cycle, but that happens considerably faster than in most similar environments, for two reasons, no, three. One, the language's grammar allows separate and highly optimized lexing, parsing, and analysis steps. Two, you can easily instruct the compiler to not generate many object files like most compilers do, and instead construct everything in memory and make only one linear commit to disk. Three, Walter Bright, the creator and original implementor of D, is an inveterate expert in optimization. This low latency means you can use D as a heck of an interpreter (the shebang notation is supported, too).

D has a true module system that supports separate compilation and generates and uses module summaries (highbrow speak for 'header files') automatically from source, so you don't need to worry about maintaining redundant files separately, unless you really wish to, in which case you can. Yep, that stops that nag right in mid-sentence.

Memory model and manycores

Given that D can call C functions directly, it may seem that D builds straight on C's memory model. That might be good news if it weren't for the pink elephant in the room dangerously close to that Ming-dynasty vase: manycores—massively parallel architectures that throw processing power at you like it's going out of style, if only you could use it. Manycores are here, and C's way of dealing with them is extremely pedestrian and error-prone. Other procedural and object-oriented languages made only little improvements, a state of affairs that marked a recrudescence of functional languages that rely on immutability to elegantly sidestep many parallelism-related problems.

Being a relative newcomer, D is in the enviable position of placing a much more informed bet when it comes to threading. And D bets the farm on a memory model that is in a certain way radically different from many others. You see, old-school threads worked like this: you call a primitive to start a new thread and the new thread instantly sees and can touch any data in the program. Optionally and with obscure OS-dependent means, a thread could also acquire the so-called thread-private data for its own use. In short, memory is by default shared across all threads. This has caused problems yesterday, and it makes today a living hell. Yesterday's problems were caused by the unwieldy nature of concurrent updates: it's very hard to properly track and synchronize things such that data stays in good shape throughout. But people were putting up with it because the notion of shared memory was a close model to the reality in hardware, and as such was efficient when gotten right. Now is where we're getting to the 'living hell' part – nowadays, memory is in fact increasingly less shared. Today's reality in hardware is that processors communicate with memory through a deep memory hierarchy, a large part of which is private to each core! So not only shared memory is hard to work with, it turns out to be quickly becoming the *slower* way of doing things because it is increasingly removed from physical reality.

Yesterday's problems were caused by the unwieldy nature of concurrent updates

While traditional languages were wrestling with all of these problems, functional languages were favoring a principled position stemming from mathematical purity: we're not interested in modeling hardware, they said, but we'd like to model true math. And math for the most part does not have mutation and is time-invariant, which makes it an ideal candidate for parallel computing. (Imagine the moment when one of those first mathematicians-turned-programmers has heard about parallel computing – they must have slapped their forehead: 'Wait a *minute*!...') It was well noted in functional programming circles that such a computational model does inherently favor out-of-order, parallel execution, but that potential was more of a latent energy than a realized goal until recent times. Today, it becomes increasingly clear that a functional, mutation-free style of writing programs will be highly relevant for at least parts of a serious application that wants to tap into parallel processing.

So where's D positioning itself in all this? There's one essential concept forming the basis of D's approach to parallelism:

> Memory is thread-private by default, shared on demand.

In D, all memory is by default private to the thread using it; even unwitting globals are allocated per-thread. When sharing is desired, objects can be qualified with `shared` which means that they are visible from several threads at once. Crucially, the type system knows about `shared` data and limits what can be done with it to ensure that proper synchronization mechanisms are used throughout. This model avoids very elegantly a host of thorny problems related to synchronization of access in default-shared threaded languages. In those languages, the type system has no idea which data is supposed to be shared and which isn't so it often relies on the honor system – the programmer is trusted to annotate shared data appropriately. Then complicated rules explain what happens in various scenarios involving unshared data, shared annotated data, data that's not annotated yet still shared, and combinations of the above – in a very clear manner so all five people who can understand them will understand them, and everybody calls it a day.

Support for manycores is a very active area of research and development, and a good model has not been found yet. Starting with the solid foundation of a default-private memory model, D is incrementally deploying amenities that don't lock its options: pure functions, lock-free primitives, good old lock-based programming, message queues (planned), and more. More advanced features such as ownership types are being discussed.

Immutability

So far so good, but what happened to all that waxing about the purity of math, immutability, and functional-style code? D acknowledges the crucial role that functional-style programming and immutability have for solid parallel programs (and not only parallel, for that matter), so it defines `immutable` as a qualifier for data that never, ever changes. At the same time D also recognizes that mutation is often the best means to a goal, not to mention the style of programming that is familiar to most of us. (If you're not among most of us: sorry, I meant 'some of us.') D's answer is rather interesting, as it encompasses mutable data and immutable data in a seamless whole.

Why is immutable data awesome? Sharing immutable data across threads never needs synchronization, and no synchronization is really the fastest synchronization around. The trick is to make sure that read-only really means read-only, otherwise all guarantees fall apart. To support that important aspect of parallel programs, D provides an unparalleled (there goes the lowest of all literary devices right there) support for functional programming. Data adorned with the `immutable` qualifier provides a strong static guarantee – a correctly typed program cannot change `immutable` data. Moreover, immutability is *deep* – if you are in immutable territory and follow a reference, you'll always stay in immutable territory. (Why? Otherwise, it all comes unglued as you think you share immutable data but end up unwittingly sharing mutable data, in which case we're back to the complicated rules we wanted to avoid in the

first place.) Entire subgraphs of the interconnected objects in a program can be 'painted' **immutable** with ease. The type system knows where they are and allows free thread-sharing for them and also optimizes their access more aggressively in single-threaded code, too.

Is D the first language to have proposed a default-private memory model? Not at all. What sets D apart is that it has integrated default-private thread memory with immutable and mutable data under one system. The temptation is high to get into more detail about that, but let's leave that for another day and continue the overview.

Safety high on the list

Being a systems-level language, D allows extremely efficient and equally dangerous constructs: it allows unmanaged pointers, manual memory management, and **cast**ing that can break into pieces the most careful design. However, D also has a simple mechanism to mark a module as 'safe,' and a corresponding compilation mode that forces memory safety. Successfully compiling code under that subset of the language – affectionately dubbed SafeD – does not guarantee you that your code is portable, that you used only sound programming practices, or that you don't need unit tests. SafeD is focussed only on eliminating memory corruption possibilities.

In SafeD you cannot have memory corruption

Safe modules (or triggering safe compilation mode) impose extra semantic checks that disallow at compilation time all dangerous language features such as forging pointers or escaping addresses of stack variables. In SafeD you cannot have memory corruption. Safe modules should form the bulk of a large application, whereas 'system' modules should be rare and far between, and also undergo increased attention during code reviews. Plenty of good applications can be written entirely in SafeD, but for something like a memory allocator you'd have to get your hands greasy. And it's great that you don't need to escape to a different language for certain parts of your application. At the time of this writing, SafeD is not finished, but is an area of active development.

Read my lips: no more axe

D is multi-paradigm, which is a pretentious way of saying that it doesn't have an axe to grind. D got the memo. Everything is not necessarily an object, a function, a list, a hashtable, or the Tooth Fairy. It depends on you what you make it. Programming in D can therefore feel liberating because when you want to solve a problem you don't need to spend time thinking of ways to adapt it to your hammer (axe?) of choice. Now, truth be told, freedom comes with responsibility: you now need to spend time figuring out what design would best fit a given problem.

By refusing to commit to One True Way, D follows the tradition started by C++, with the advantage that D provides more support for each paradigm in turn, better integration between various paradigms, and considerably less friction in following any and all of them. This is the advantage of a good pupil; obviously D owes a lot to C++, as well as less eclectic languages such as Java, Haskell, Eiffel, Javascript, Python, and Lisp. (Actually most languages owe their diction to Lisp, some just don't admit it.)

Object-oriented features

In D you get **struct**s and then you get **class**es. They share many amenities but have different charters: **struct**s are value types, whereas **class**es are meant for dynamic polymorphism and are accessed solely by reference. That way confusions, slicing-related bugs, and comments à la `// No! Do NOT inherit!` do not exist. When you design a type, you decide upfront whether it'll be a monomorphic value or a polymorphic reference. C++ famously allows defining ambiguous-gender types, but their use is rare, error-prone, and objectionable enough to warrant simply avoiding them by design.

D's object-orientation offering is similar to Java's and C#'s: single inheritance of implementation, multiple inheritance of interface. In doing so, D doesn't go with the sour-grapes theory 'Multiple Inheritance is Evil: How an Amulet Can Help'. Instead, D simply acknowledges the difficulty in making multiple inheritance of state work efficiently and in useful ways. To provide most of the benefits of multiple inheritance at controllable cost, D allows a type to use multiple *subtyping* like this:

```
class WidgetBase { ... }
class Gadget { ... }
class Widget : WidgetBase, Interface1, Interface2
{
  Gadget getGadget() { ... }
  alias getGadget this; // Widget subtypes
                        // Gadget!
}
```

The **alias** introduction works like this: whenever a **Gadget** is expected but all you have is a **Widget**, the compiler calls **getGadget** and obtains it. The call is entirely transparent, because if it weren't, that wouldn't be subtyping; it would be something frustratingly close to it. (If you felt that was an innuendo, it probably was.) Moreover, **getGadget** has complete discretion over completing the task—it may return e.g. a sub-object of **this** or a brand new object. You'd still need to do some routing to intercept method calls if you need to, which sounds like a lot of boilerplate coding, but here's where D's reflection and code generation abilities come to fore (see below). The basic idea is that D allows you to subtype as you need via **alias this**. You can even subtype **int** if you feel like it.

D has integrated other tried and true techniques from experience with object orientation, such as an explicit **override** keyword to avoid accidental overriding, direct support for signals and slots, and a technique I can't mention because it's trademarked, so let's call it contract programming.

Functional programming

Quick, how do you define a functional-style Fibonacci function?

```
uint fib(uint n)
{
  return n < 2 ? n : fib(n -1) + fib(n -2);
}
```

I confess to entertaining fantasies. One of these fantasies has it that I go back in time and somehow eradicate this implementation of Fibonacci such that no Computer Science teacher ever teaches it. (Next on the list are bubble sort and the *O(n log n)-space* quicksort implementation. But **fib** outdoes both by a large margin. Also, killing Hitler or Stalin is dicey as it has hard-to-assess consequences, whereas killing **fib** is just good.) **fib** takes *exponential* time to complete and as such promotes nothing but ignorance of complexity and of the costs of computation, a 'cute excuses sloppy' attitude, and SUV driving. You know how bad exponential is? **fib(10)** and **fib(20)** take negligible time on my machine, whereas **fib(50)** takes nineteen and a half minutes. In all likelihood, evaluating **fib(1000)** will outlast humankind, which gives me solace because it's what we deserve if we continue teaching it.

Fine, so then what does a 'green' functional Fibonacci implementation look like?

```
uint fib(uint n)
{
  uint iter(uint i, uint fib_1, uint fib_2)
  {
    return i == n
      ? fib_2
      : iter(i + 1, fib_1 + fib_2, fib_1);
  }
  return iter(0, 1, 0);
}
```

The revised version takes negligible time to complete **fib(50)**. The implementation now takes *O(n)* time, and tail call optimization (which D implements too) takes care of the space complexity. The problem is that

that's how D defines functional purity: you can use mutation in the implementation of a pure function, as long as it's transitory and private

the new **fib** kind of lost its glory. Essentially the revised implementation maintains two state variables in the disguise of function parameters, so we might as well come clean and write the straight loop that **iter** made unnecessarily obscure:

```
uint fib(uint n)
{
  uint fib_1 = 1, fib_2 = 0;
  foreach (i; 0 .. n)
  {
    auto t = fib_1;
    fib_1 += fib_2;
    fib_2 = t;
  }
  return fib_2;
}
```

but (shriek of horror) this is not functional anymore! Look at all that disgusting mutation going on in the loop! One mistaken step, and we fell all the way from the peaks of mathematical purity down to the unsophisticatedness of the unwashed masses.

But if we sit for a minute and think of it, the iterative **fib** is not *that* unwashed. If you think of it as a black box, **fib** always outputs the same thing for a given input, and after all pure is what pure does. The fact that it uses private state may make it less functional in letter, but not in spirit. Pulling carefully on that thread, we reach a very interesting conclusion: as long as the mutable state in a function is entirely *transitory* (i.e., allocated on the stack) and *private* (i.e., not passed along by reference to functions that may taint it), then the function can be considered pure.

And that's how D defines functional purity: you can use mutation in the implementation of a pure function, as long as it's transitory and private. You can then put **pure** in that function's signature and the compiler will compile it without a hitch:

```
pure uint fib(uint n)
{
  ... iterative implementation ...
}
```

The way D relaxes purity is pretty cool because you're getting the best of both worlds: iron-clad functional purity guarantees, and comfortable implementation when iteration is the preferred method. If that's not cool, I don't know what is.

Last but not least, functional languages have another way of defining a Fibonacci sequence: as a so-called infinite list. Instead of a function, you define a lazy infinite list that gives you more Fibonacci numbers the more you read from it. D offers a pretty cool way of defining such lazy lists. For example, the code below outputs the first 50 Fibonacci numbers (you'd need to import **std.range**):

```
foreach (f; take(50,
   recurrence!("a[n-1] + a[n-2]")(0, 1)))
{
  writeln(f);
}
```

That's not a one-liner, it's a half-liner! The invocation of **recurrence** means, create an infinite list with the recurrence formula **a[n] = a[n-1] + a[n-2]** starting with numbers 0 and 1. In all this there is no dynamic memory allocation, no indirect function invocation, and no non-renewable resources used. The code is pretty much equivalent to the loop in the iterative implementation. To see how that can be done, you may want to read through the next section.

Generic programming

(You know the kind of caution you feel when you want to describe to a friend a movie, a book, or some music you really like but don't want to spoil by overselling? That's the kind of caution I am feeling as I'm approaching the subject of generic programming in D.) Generic programming has several definitions—even the neutrality of the Wikipedia article on it is being disputed at the time of this writing. Some people refer to generic programming as 'programming with parameterized types a.k.a. templates or generics', whereas others mean 'expressing algorithms in the most generic form that preserves their complexity guarantees'. I'll discuss a bit of the former in this section, and a bit of the latter in the next section.

D offers parameterized **structs**, **class**es, and functions with a simple syntax, for example here's a **min** function:

```
auto min(T)(T a, T b) { return b < a ? b : a; }
...
auto x = min(4, 5);
```

T would be a type parameter and **a** and **b** are regular function parameters. The **auto** return type leaves it to the compiler to figure out what type **min** returns. Here's the embryo of a list:

```
class List(T)
{
  T value;
  List next;
  ...
}
...
List!int numbers;
```

The fun only starts here. There's too much to tell in a short article to do the subject justice, so the next few paragraphs offer 'deltas' – differences from the languages with generics that you might already know.

Parameter kinds

Not only types are acceptable as generic parameters, but also numbers (integral and floating-point), strings, compile-time values of **struct** type, and *aliases*. An alias parameter is any symbolic entity in a program, which can in turn refer to a value, a type, a function, or even a template. (That's how D elegantly sidesteps the infinite regression of template template template ... parameters; just pass it as an alias.) Alias parameters are also instrumental in defining lambda functions. Variable length parameter lists are also allowed.

String manipulation

Passing strings to templates would be next to useless if there was no meaningful static manipulation of strings. D offers full string manipulation capabilities during compilation (concatenation, indexing, selecting a substring, iterating, comparison. . .).

Code generation: the assembler of generic programming

Manipulating strings during compilation may be interesting, but is confined to the data flatland. What takes things into space is the ability to convert strings into code (by use of the **mixin** expression). Remember the **recurrence** example? It passed the **recurrence** formula for Fibonacci sequences into a library facility by means of a string. That facility in turn converted the string into code and provided arguments to it. As another example, Listing 1 shows how you sort ranges in D.

Code generation is very powerful because it allows implementing entire features without a need for language-level support. For example, D lacks bit fields, but the standard module **std.bitmanip** defines a facility implementing them fully and efficiently.

Listing 1

```
// define an array of integers
auto arr = [ 1, 3, 5, 2 ];
// sort increasingly (default)
sort(arr);
// decreasingly, using a lambda
sort!((x, y) { return x > y; })(arr);
// decreasingly, using code generation;
// comparison is a string with conventional
// parameters a and b
sort!("a > b")(arr);
```

Introspection

In a way, introspection (i.e., the ability to inspect a code entity) is the complement of code generation because it looks at code instead of generating it. It also offers support for code generation—for example, introspection provides the information for generating a parsing function for some enumerated value. At this time, introspection is only partially supported. A better design has been blueprinted and the implementation is 'on the list', so please stay tuned for more about that.

is and static if

Anyone who's written a nontrivial C++ template knows both the necessity and the encumbrances of (a) figuring out whether some code 'would compile' and deciding what to do if yes vs. if not, and (b) checking for Boolean conditions statically and compiling in different code on each branch. In D, the Boolean compile-time expression `is(typeof(expr))` yields `true` if `expr` is a valid expression, and `false` otherwise (without aborting compilation). Also, `static if` looks pretty much like `if`, except it operates during compilation on any valid D compile-time Boolean expression (i.e., `#if` done right). I can easily say these two features alone slash the complexity of generic code in half, and it filled me with chagrin that C++0x includes neither.

But wait, there's. . .well, you know

Generic programming is a vast play field, and although D covers it with a surprisingly compact conceptual package, it would be hard to discuss matters further without giving more involved information. D has more to offer, such as customized error messages, constrained templates à la C++0x concepts, tuples, a unique feature called local instantiation (crucial for flexible and efficient lambdas), and, if you call within the next five minutes, a knife that can cut through a frozen can of soda.

ever since the STL appeared, the landscape of containers+ algorithms libraries has forever changed

A word on the standard library

This subject is a bit sensitive politically because, as mentioned, there are two full-fledged libraries that can be used with D, Phobos and Tango. I only worked on the former so I will limit my comments to it.

For my money, ever since the STL appeared, the landscape of containers+ algorithms libraries has forever changed. It's changed so much, in fact, that every similar library developed after the STL but in ignorance of it runs serious risks of looking goofy. (Consequently, a bit of advice I'd give a programmer in any language is to understand the STL.) This is not because STL would be a perfect library – it isn't. It is inextricably tied to the strengths and weaknesses of C++, for example it's efficient but it has poor support for higher-order programming. Its symbiosis with C++ also makes it difficult for non-C++ programmers to understand the STL in abstract, because it's hard to see its essence through all the nuts and bolts. Furthermore, STL has its own faults, for example its conceptual framework fails to properly capture a host of containers and ways of iterating them.

STL's main merit was to reframe the entire question of what it means to write a library of fundamental containers and algorithms, and to redefine the process of writing one in wake of the answer. The question STL asked was: 'What's the minimum an algorithm could ever ask from the topology of the data it's operating on?' Surprisingly, most library implementers and even some algorithm pundits were treating the topic without due rigor. STL's angle put it in stark contrast to a unifying interface view in which, for example, it's okay to unify indexed access in arrays and linked lists because the topological aspect of performing it can be written off as just an implementation detail. STL revealed the demerit of such an approach because, for example, it's disingenuous to implement as little as a linear search by using a unifying interface (unless you enjoy waiting for quadratic algorithms to terminate). These are well-known truths to anyone serious in the least about algorithms, but somehow there was a disconnect between understanding of algorithms and their most general implementations in a programming language. Although I was conversant with algorithm fundamentals, I can now say I had never really thought of what the pure, quintessential, Platonic linear search is about until I first saw it in the STL fifteen years ago.

a system-level language without the agonizing pain, an application language without the boredom, a principled language without the attitude

That's a roundabout way of saying that Phobos (places to look at in the online documentation: `std.algorithm` and `std.range`) is a lot about the STL. If you ask me, Phobos' offering of algorithms is considerably better than STL's, and for two reasons. One, Phobos has the obvious advantage of climbing on the shoulders of giants (not to mention the toes of dwarfs). Two, it uses a superior language to its fullest.

Ranges are hot, iterators are not

Probably the most visible departure from the STL is that there are no iterators in Phobos. The iterator abstraction is replaced with a range abstraction that is just as efficient but offers vastly better encapsulation, verifiability, and abstraction power. (If you think about it, *none* of an iterator's primitive operations are naturally checkable. That's just bizarre.) Code using ranges is as fast, safer, and terser than code using iterators – no more `for` loop that's too long to contain in one line. In fact, thinking and coding with ranges is so much terser, new idioms emerge that may be thinkable but are way too cumbersome to carry through with iterators. For example, you might have thought of a `chain` function that iterates two sequences one after another. Very useful. But `chain` with iterators takes four iterators and returns two, which makes it too ham-fisted to be of any use. In contrast, `chain` with ranges takes two ranges and returns one. Furthermore, you can use variadic arguments to have `chain` accept any number of ranges – and all of a sudden, we can avail ourselves of a very useful function. `chain` is actually implemented in the standard module `std.range`. As an example, here's how you can iterate through three arrays:

```
int[] a, b, c;
...
foreach (e; chain(a, b, c))
{
... use e ...
}
```

Note that the arrays are not concatenated! `chain` leaves them in place and only crawls them in sequence. This means that you might think you could

Hunting the Snark (Part 2)

Alan Lenton continues his job hunt.

'I can see what the candidate has done in the past from looking at their CV. What I really want to know is what they are capable of doing in the future.'

The above quote is from a CTO in a discussion I had with him on interviewing techniques, after I started working for him. I think it goes straight to the nub of the problem. Given the rapid change in technologies in the IT business, past 'performance' is not always a sure guide to future ability.

It seems to me that when it comes to job interviews we can identify the following elements that need to be considered by the interviewers:

1. What do we want out of the interview?
2. Who is going to do the interview?
3. Advance preparation. What are we going to talk about with the candidate that will elicit the information identified in item 1?
4. Environment: Is the interview going to be carried out in a working environment? If not, what is the justification for the environment chosen?

Interviewees need to consider their normal working practices, and how they are going to translate those practices into something convincing in an interview environment.

Finding a goal – preferably not an own goal

We'll start with a look at the interviewing side of a job. The first question is what do we want out of the interview? The answer is obvious. Somebody to do the job. Unfortunately, that's where most people leave it, perhaps with a wave of the job advert, which is usually a boiler-plate shopping list lifted from someone else's advert.

ALAN LENTON

Alan is a programmer, a sociologist, a games designer, a wargamer, writer of a weekly tech news and analysis column, and an occasional writer of short stories. None of these skills seem to be appreciated by putative employers...

The Case for D (continued)

change elements in the original arrays by means of **chain**, which is entirely true. For example guess what this code does:

```
sort(chain(a, b, c));
```

You're right – the collective contents of the three arrays has been sorted, and, without modifying the size of the arrays, the elements have been efficiently arranged such that the smallest come in a and so on. This is just a small example of the possibilities offered by ranges and range combinators in conjunction with algorithms.

Laziness to infinity and beyond

STL algorithms (and many others) are eager: by the time they return, they've finished their job. In contrast, Phobos uses lazy evaluation whenever it makes sense. By doing so, Phobos acquires better composition capabilities and the ability to operate with infinite ranges. For example, consider the prototypical higher-order function **map** (popular in functional programming circles, not to be confused with the homonym STL data structure) that applies a given function to each element in a range. If **map** were insisting to be eager, there'd be two problems. First, it would have to allocate new space to store the result (e.g., a list or an array). Second, it would have to consume the range in its entirety before returning control to the caller. The first is an efficiency problem: memory allocation could and should be avoided in many cases (for example, the caller wants to just look at each result of **map** in turn). The second is a problem of principles: eager **map** simply can't deal with infinite ranges because it would loop forever.

That's why Phobos defines **map** to return a lazy range – it incrementally makes progress as you consume elements from it. In contrast, the **reduce** function (in a way a converse of **map**) is eager. Some functions need both lazy and eager versions. For example, **retro(r)** returns a range that iterates the given range **r** backwards, whereas **reverse(r)** reverses **r** in place.

Conclusion

There would be more things to talk about even in an overview, such as unit tests, UTF strings, compile-time function evaluation (a sort of D interpreter running during compilation of a D program), dynamic closures, and many others. But with any luck, your curiosity has been piqued. If you are looking for a system-level language without the agonizing pain, an application language without the boredom, a principled language without the attitude, or – most importantly – a weighted combination thereof, then D may be for you.

If you feel like asking further questions, write the author, or better yet, tune to the Usenet server news.digitalmars.com and post to the digitalmars.d newsgroup – the hub of a vibrant community. ■

If you don't have a worked out, and focussed, view of what the job involves, how you expect it to evolve, and what skills are going to be needed both now and in the future, then your chances of finding someone suitable are less than if you had made a random pick.

A nice bedside manner

The next question – who is going to be doing the interviewing? – is not really very often considered. Usually people are just drafted in to make up the panel, and left to get on with it.

Think about it though. Why are people from a profession notorious for its lack of social skills allowed to carry out recruitment interviews? Their decisions may well have a serious impact on the future of the business. And why is no one ever given any training on how to carry out an interview?

A starter for ten...

OK – so we have assembled a (usually very homogenous) group of people who are going to do the interview. Now comes the question of what to ask. This is a thorny issue indeed. Are you looking for a specialist or for a generalist? The questions will be completely different. How does your shop work – what sort of questions will get you that sort of information?

Of course, you don't actually have to ask questions at all. Some of the most effective interviews I've been to had no questions per se, they were a series of discussions based on my CV. The discussions covered the work I'd done, how and why I made the decisions I did, and how I would use that experience to solve related problems in other situations.

In general, there seem to be two ways to carry out an interview. Either you can opt to find out what the candidate doesn't know, or you can try to find out what they do know. Given the huge scope of C++, and the varied working practices, the chances of anyone, even Bjarne, knowing everything are close to zero.

Obviously you need to know in general terms in what areas the candidate has gaps in their experience. Having found a gap, all you really want to know is whether the candidate has made any attempt to fill in the gap by reading, discussion with co-workers, etc. That said, there is little point in pursuing the details in this situation – it just ends up making the candidate miserable and hesitant in answering questions about the things they do understand.

If you do find that the candidate has a gap in their knowledge in an area that you would have expected them to know, then the most important thing is to find out why. They may, of course, have been lying on their CV, but that isn't always the case. It may well be that there are perfectly good reasons why there is a gap. Perhaps, for instance, the way they design programs is such that they have never needed that feature of the language.

I once went to an interview for a Qt programmer in which one of the interviewers was totally focussed on the `qmake` tool (used to produce intermediate files when using Qt libraries). I've never used the tool, though I've been using Qt for several years. I use Qt's Visual Studio plug-in which handles everything via VS's solution/project system.

By the end of the interview I was starting to get really fed up with the mantra, 'Let's go back to qmake'. He spent so much time 'proving' that I didn't know much about `qmake` (something I explained right at the start), that he never did find out what I knew about Qt programming.

The whiteboard as a weapon

And finally, what about the interview environment...

Let me start by asking a question. Would you expect someone to take the practical driving test by explaining, using a whiteboard, how they would cope with different situations? Details to include what they are doing with their hands, feet, eyes and ears...

Of course you wouldn't. But we do expect people to solve programming problems without a computer in an interview.

Have you ever tried to drive while thinking what your feet are doing? That's very similar to the problems you pose when you ask a syntax question sitting across a table in an interview. The candidate has probably typed it into the computer thousands of times, automatically, and doesn't even think about it any more. It's the computing equivalent of changing gear in a manual car!

What do you really want?

Be careful what you ask for, you might just get it. You could easily end up with someone who can do exactly what you want at this moment in time, but is incapable of moving on to the things you want to do next year.

Probably the most difficult thing to find out is whether a candidate is able to learn and move forward.

Many years ago I went for a job as a London bus driver. The 'interview' consisted of driving a double decker bus round London in the rush hour! It went something like this: the instructor drove the bus for 10–15 minutes and gave me a running commentary on what he was doing. We then swapped over and I drove for 15 minutes (I'd never driven a bus before, it was terrifying).

We pulled into the side of the road, and the instructor went through what I'd done wrong (mostly not taking corners wide enough and bumping over the pavement) and what was the best way to correct the problems. Then I got back in the hot seat and drove around for another 20 minutes.

It was that last 20 minutes that was the test. I was being watched to see how well I corrected the errors. The instructor already knew I could drive, because I had a licence. What he wanted to know was could I be taught to drive a bus. Fortunately for the programming world, although I passed that test, my eyesight wasn't up to their stringent standards, and I was turned down. So I had to resort to the much less glamorous profession of programming instead!

I don't know what the equivalent of that test is for programming, I wish I did, it would make life much easier, both for the interviewers and the interviewees. There was an extended discussion on interviews on the accu-general list recently which is worth reading and presents differing views on the topic. I'd recommend a read of it should you end up having to act as an interviewer (search the archives for the subject 'Torturing interviewees' – yes, really).

Next issue I'll take a look at this problem from the interviewee's point of view, but in the mean time I'll leave you with the Cheshire Cat's take on things:

'Would you tell me, please, which way I ought to go from here?"

'That depends a good deal on where you want to get to,' said the Cat.

'I don't much care where –' said Alice.

'Then it doesn't matter which way you go,' said the Cat.

'– so long as I get somewhere,' Alice added as an explanation.

'Oh, you're sure to do that,' said the Cat, 'if you only walk long enough.'

From Alice in Wonderland, by Lewis Carrol ■

Code Critique Competition 57

Set and collated by Roger Orr.

Please note that participation in this competition is open to all members, whether novice or expert. A book prize is awarded for the best entry. Readers are also encouraged to comment on published entries, and to supply their own possible code samples for the competition (in any common programming language) to scc@accu.org.

Last issue's code

I'm writing a simple hash-map and I've got a couple of questions. Firstly, why does the line marked '1' in the header file need the word 'typename'? Secondly, my little test program doesn't quite work – the last line of output still shows the old value for 'key2'.

Listing 1

```
// ---- hash.h -----
#include <algorithm>
#include <list>
#include <numeric>
#include <vector>
template <class Key, class Value>
class hashmap
{
public:
  hashmap( int size = 10 )
  : buckets(size) {}
  void set( Key key, Value value );
  Value get( Key key );
private:
  struct entry
  {
    Key key;
    Value value;
    bool operator==( Value key ) const
    {
      return key == key;
    }
  };
  int hashfun( Key s );
  std::vector< std::list<entry> > buckets;
  std::list<entry> bucket;
  typename std::list<entry>::iterator it; // 1
};
template <class Key, class Value>
void hashmap<Key, Value>::set( Key key,
  Value value )
{
  int hashval = hashfun( key );
  int thebucket = hashval % buckets.size();
  bucket = buckets[ thebucket ];
  if ( bucket.size() == 0 )
  {
    entry e;
    e.key = key;
    e.value = value;
    buckets[ thebucket ].push_back( e );
  }
  else
  {
    it = std::find( bucket.begin(),
      bucket.end(), key );
    if ( it == bucket.end() )
    {
      entry e;
      e.key = key;
      e.value = value;
      bucket.push_back( e );
    }
    else
    {
      it->value = value;
    }
  }
}
template <class Key, class Value>
Value hashmap<Key, Value>::get( Key key )
{
  int hashval = hashfun( key );
  int thebucket = hashval % buckets.size();
  bucket = buckets[ thebucket ];
  it = std::find( bucket.begin(),
    bucket.end(), key );
  if ( it == bucket.end() )
  {
    return Value();
  }
  else
  {
    return it->value;
  }
}
template <class Key, class Value>
int hashmap<Key, Value>::hashfun( Key s )
{
  return std::accumulate( s.begin(),
    s.end(), 0 );
}
// ---- hash.cpp -----
#include <iostream>
#include <string>
#include "hash.h"
int main()
{
  hashmap<std::string, std::string> test;
  test.set("key1", "value1");
  test.set("key2", "value2");
  std::cout << "key1 = "
    << test.get( "key1" ) << std::endl;
  std::cout << "key2 = "
    << test.get( "key2" ) << std::endl;
  std::cout << "key3 = "
    << test.get( "key3" ) << std::endl;
  test.set("key2", "another_value2");
  std::cout << "key2 = "
    << test.get( "key2" ) << std::endl;
}
```

Listing 1 (cont'd)

ROGER ORR

Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002.
He may be contacted at rogero@howzatt.demon.co.uk

Critiques

David Pol <david@kalarelia.com>

First of all, why does the line marked '1' in the header file need the keyword **typename**? It does need it because entry is a dependent name, that is, a name that depends on a template parameter. The types referred to by dependent names are called dependent types. In particular, a nested class in a dependent class template is a dependent type, which is precisely what we have in our code. And all dependent types need the keyword **typename** before them, because the C++ parsing rules state that dependent names should be parsed as non-types (even if that leads to a syntax error). The **typename** keyword is a way of telling the compiler we want the name that follows to be treated as a type.

Apparently, our code is failing when setting the value for a key that is already stored in the hashmap. Let's start by debugging the process whereby a new key-value pair is inserted into the hashmap (i.e., the code that belongs to the **hashmap<>::set()** function).

```
int hashval = hashfun(key);
int thebucket = hashval % buckets.size();
```

This computes the hash value that corresponds to the given key, which is then used to get an index into the **std::vector<>** of buckets. So far, so good (implementation of **hashmap<>::hashfun()** is not ideal, though; it could be enhanced by passing a custom accumulator to **std::accumulate()** that performs a more sophisticated form of hashing).

```
bucket = buckets[thebucket];
```

The member variable **bucket** is declared as a **std::list<entry>**. This line assigns to **bucket** a copy (a potentially expensive copy, by the way!) of the **std::list<entry>** element located at index **thebucket** in buckets. And, in the situation where **bucket.size() > 0**, we are actually modifying this local copy, and not the original bucket. That's the main problem with this code: we never modify buckets.

In fact, the **bucket** member variable is not necessary at all. We can replace it with a **std::list<entry>&** local to the **hashmap<>::get()** and **hashmap<>::set()** functions. Reusing the same variable makes the code less clear and concise, and adds unnecessary overhead to the **hashmap<>** class. Exactly the same can be said about the **it** member variable.

Although the current standard does not provide hash table-based containers (something C++0x will fix, fortunately), several alternatives are available, ranging from compiler-specific extensions (e.g. **stdext** namespace in MVC++ or **__gnu_cxx** namespace in gcc) to portable implementations of the next standard (e.g. Boost.Unordered). This is also a good example of why making use of existing solid libraries is usually the way to go; consider the code we get and the problems we avoid by using **Boost.Unordered**:

```
#include <iostream>
#include <ostream>
#include <string>
#include <boost/unordered_map.hpp>
int main()
{
  boost::unordered_map
    <std::string, std::string> test;
  test["key1"] = "value1";
  test["key2"] = "value2";
  std::cout << "key1 = " << test["key1"]
    << std::endl;
  std::cout << "key2 = " << test["key2"]
    << std::endl;
  std::cout << "key3 = " << test["key3"]
    << std::endl;
  test["key2"] = "another_value2";
  std::cout << "key2 = " << test["key2"]
    << std::endl;
}
```

Also, there are three points that are worth mentioning with regards to our original code:

- There is a bug in **entry::operator==()**. The type of the parameter key should be **Key**, not **Value**. If we try to use a **hashmap<>** that has different types for the keys and the values, we will get a compilation error. Also, it's good to use a **const** reference for it to avoid unnecessary copying (the same goes for the parameters in all the member functions of **hashmap**).
- The header file does not make use of include guards to protect against multiple inclusion.
- The constructor for **hashmap** can be used an implicit conversion function, which is generally undesired. We can mark it as **explicit** to protect against such silent conversions.

Our final code for the hashmap, then, looks like this:

```
#ifndef HASH_MAP_INCLUDED_H
#define HASH_MAP_INCLUDED_H
#include <algorithm>
#include <list>
#include <numeric>
#include <vector>
#include <iostream>
template <class Key, class Value>
class hashmap
{
public:
  explicit hashmap(std::size_t size = 10)
  : buckets(size) {}
  void set(const Key& key,
    const Value& value);
  Value get(const Key& key);
private:
  struct entry
  {
    Key key;
    Value value;
    bool operator==(const Key& rhs) const
    {
      return key == rhs;
    }
  };
  int hashfun(const Key& s);
  std::vector<std::list<entry> > buckets;
};
template <class Key, class Value>
void hashmap<Key, Value>::set(const Key& key,
  const Value& value) {
  int hashval = hashfun(key);
  int thebucket = hashval % buckets.size();
  std::list<entry>& bucket
    = buckets[thebucket];
  if (bucket.size() == 0)
  {
    entry e;
    e.key = key;
    e.value = value;
    bucket.push_back(e);
  }
  else
  {
    typename std::list<entry>::iterator it
      = std::find(bucket.begin(),
                  bucket.end(), key);
    if (it == bucket.end())
    {
      entry e;
      e.key = key;
      e.value = value;
```

```
      bucket.push_back(e);
    }
    else
    {
      it->value = value;
    }
  }
}
template <class Key, class Value>
Value hashmap<Key, Value>::get(const Key& key)
{
  int hashval = hashfun(key);
  int thebucket = hashval % buckets.size();
  std::list<entry>& bucket
    = buckets[thebucket];
  typename std::list<entry>::iterator it
    = std::find(bucket.begin(),
                bucket.end(), key);
  if (it == bucket.end())
  {
    return Value();
  }
  else
  {
    return it->value;
  }
}
template <class Key, class Value>
```

Listing 2

```
#include <stdio.h>
#include <string.h>
typedef enum {
  Hearts,
  Diamonds,
  Clubs,
  Spades
} Suit;
typedef struct Card
{
    Suit suit;
    int value;
} Card;
typedef Card Deck[52];
void LoadDeck(Deck * myDeck)
{
  int i = 0;
  for(; i < 51; i++)
  {
    myDeck[i]->suit = i % 4;
    myDeck[i]->value = i % 13;
  }
}
void PrintDeck(Deck * myDeck)
{
  int i = 0;
  for(;i < 52; i++)
  {
     printf("Card %d %d\n", myDeck[i]->suit,
       myDeck[i]->value);
  }
}
int main()
{
  Deck myDeck;
  memset(&myDeck,0,sizeof(Deck));
  LoadDeck(&myDeck);
  PrintDeck(&myDeck);
  return 0;
}
```

```
int hashmap<Key, Value>::hashfun(const Key& s)
{
  return std::accumulate(s.begin(),
    s.end(), 0);
}
#endif // HASH_MAP_INCLUDED_H
```

Commentary

The code looked to me like a case of premature generalisation. As David pointed out, the equality operator of the **hashmap::entry** was incorrectly defined to take an argument of the **Value** type although it needed an argument of the **Key** type.

It is usually simpler for most programmers to write a non-templated class first and then generalise it, if necessary, once it is working correctly. Doing so also delays the need to use **typename**!

A couple of general points about the template header file. The first is that the header file includes the implementation of the class as well as its definition. Consequently there is more clutter in the header file so making it more important to try and highlight the important items for the reader. Additionally, as the implementation must be included this also means additional header files are required which makes the resultant header file more bigger and potentially slower to compile.

I would also want to draw attention to the hash function used. The performance of hashed containers is critically dependent on the size of the hash table and the implementation of the hash function. Simply getting the code to compile and execute correctly is not enough – in the worst case a hashed container can degenerate into a single linked list which is slower to access than a standard map.

The main characteristic desired for the hash function is that it will avoid collisions between inputs (that is where two different inputs generate the same output). Consider this hash function when used, for instance, against all two letter English words. There are 101 of these (depending on precisely how you count them) but the algorithm used only produces 37 different hash values ranging between 130 and 173. So if we picked a hash bucket size of 100 every word would be in a bucket between 30 and 73 and the other buckets would be empty.

Fortunately we can use the work of others – by using a third party hash table or the **unordered_map** in TR1 (soon to be in C++0x) we can make use of better hash functions that this one without needing to become an expert in the topic ourselves.

The Winner of CC 56

Once again there was only one entrant, which is a shame – although I suppose it does make the task of judging the entries easier.

I think David covered the ground well, and hopefully the putative writer of the code would not only have had their problem solved but also have gained a clearer understanding of templates.

Code Critique 57

(Submissions to scc@accu.org by June 1st)

Can someone please help me to understand why the following trivial C program crashes?

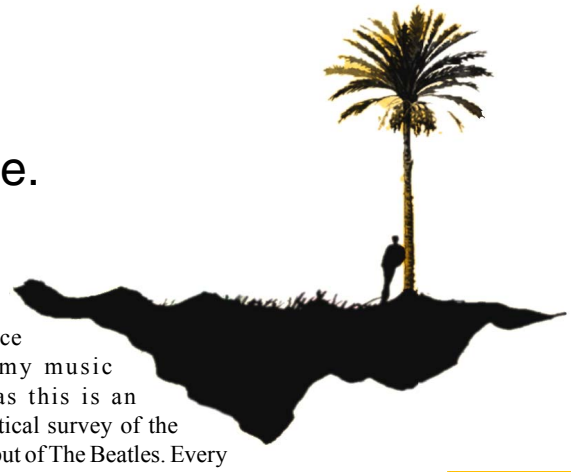
Note: You may answer the question in C, or convert the program to C++ (with justification). The code is shown in Listing 2, and is based on a posting to news://alt.comp.lang.learn.c-c++

You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/).

This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Desert Island Books

Paul Grenyer introduces Ewan Milne.

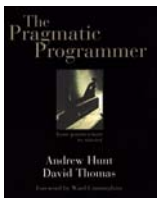


Ewan is another of the strong and active characters I encountered on accu-general, and went on to meet at a conference, soon after I joined the ACCU. In his time Ewan has made an excellent job of both ACCU and conference chairman. Whenever I think of Ewan I always remember the night I met his wife at a conference dinner. I've never been allowed to forget that, for some reason, the words "Ah! You must be Mrs Ewan", like so many other things, seemed like a good idea at the time...

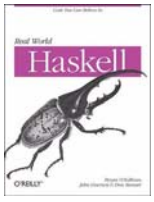
Ewan Milne

So it turns out that there's a pretty high correlation between ACCU membership and getting lost at sea. We probably want to keep this quiet lest all our travel insurance premiums go through the roof. A previous castaway wondered if we are all stranded on the same island: I have decided that we are, but one after the other. The plus point of this is that my predecessors were all so excited on their rescue that they left their books and records behind, which means that I have inherited a very respectable technical library even before I make my choices. As payback for my high-handedness, with so many classic texts already available for my browsing pleasure, what other titles am I going to choose?

For my first choice, I'm going back to the very start of my career. Although I had dabbled in home computing while at school, it was when I first opened the pages of *The C Programming Language* by Brian Kernighan and Dennis Ritchie1 in my first year at uni that I really got it. Oddly, while the preface says that the book 'is not an introductory programming manual; it assumes some familiarity with basic programming concepts', I recall this book fulfilling pretty much that role. The clarity and precision with which the concepts are presented means that not only is the language itself explained in an admirably concise 228 pages, it also inspired in me a lasting passion for programming. And I do believe this is the text that introduced `"hello, world"` to the, err, world.



The elegance of K&R's prose sets a standard that few authors can match, and many technical books are at best functional (that's with a small 'f': I'll get on to the other sort in a minute). Occasionally a book comes along that does really grab you, and for me one of the most enjoyable to return to is *The Pragmatic Programmer* by Andrew Hunt and David Thomas. There are several books that cover similar ground, the overall craft of software design and construction; many of them also of great merit. But this is the one that stands out for me, because the underlying philosophy immediately struck a chord.



My next choice is a book I haven't read yet. In fact, I just checked on Amazon and, at the time of writing, it is still to be published in the UK. But a couple of months on a desert island sounds like the ideal opportunity to devote enough time to learning something new. A few survival skills might be a more sensible option, but I'm going to take the chance to properly learn a Functional language, and so I'm taking a punt on *Real World Haskell* by Bryan O'Sullivan, John Goerzen and Donald Stewart. Simon Peyton Jones gives it his seal of approval ('Best of all, this book will expand your mind'), and judging by the table of contents, it covers monads in depth: a topic that Simon's keynote at last year's conference, entertaining and illuminating as it was, left somewhat unclear in my mind.

My last non-fiction selection is a change of pace, and a book I have enjoyed browsing regularly for many years: *Revolution In The Head* by Ian MacDonald. One reason for this choice is to ease my music selection, as this is an extensive critical survey of the recorded output of The Beatles. Every single song released by the group is individually catalogued and dissected in chronological order of its recording. Although this is undeniably a book for Beatles obsessives, it manages to avoid being either dryly academic [2] or overly sycophantic. The Fabs musical output is discussed within the context of the rapid social changes of the 1960s, and the analysis is lively and insightful. MacDonald is not afraid to point out the flaws in the Beatles canon, and his critiques of many songs reveal subtleties that may easily be missed by the casual listener. In the absence of any Beatles records, this will make a fine substitute.

Finally for reading material, a novel. I have never been big on re-reading novels, but if stranded far from home on a dry, dusty desert island, an old favourite to remind me of home would be *The Crow Road* by Iain Banks. When I first read this, I was of a similar age to the lead character – a student returning home to Argyll and his extended and eccentric family. It would be interesting to revisit it 15 years on. It starts with one of the great opening lines in fiction – if a little self-conscious ('It was the day my grandmother exploded'), and develops from there into an rambling exploration of dark family secrets, Scottishness, religion versus atheism, and the perfect way to microwave a poppadom.

So now things get tough. Choosing two records to take with me is a real struggle. Like many, I like to think that I have a varied, highly catholic musical taste – but while it is true that the CD collection at home ranges from world music to modern classical, with the odd diversion into jazz and country, the truth is that the majority of the music I listen to is made by pale skinny guys with guitars. And while I say is made, in general was made would be more accurate – in the 1980s or earlier. Still, I was stumped for some time, and had a seriously overlong short list until I imposed on myself a limitation. If any band or artist had more than one album on the list, they all had to go. So, much as it pained me, out went REM, Leonard Cohen, The Waterboys, Tom Waits, The Triffids, Joy Division, Blue Aeroplanes, David Bowie, The Smiths, The Go-Betweens, Neil Young and Radiohead. And several others.

What's it all about?

Desert Island Disks is one of BBC Radio 4's most popular and enduring programmes:

http://www.bbc.co.uk/radio4/factual/desertislanddiscs.shtml

The format is simple: each week a guest is invited to choose the eight records they would take with them to a desert island.

I've been thinking for a while that it would be entertaining to get ACCU members to choose their Desert Island Books. The format will be slightly different from the Radio 4 show. Members will choose about 5 books, one of which must be a novel, and up to two albums. The programming books must have made a big impact on their programming life or be ones that they would take to a desert island. The inclusion of a novel and a couple of albums will also help us to learn a little more about the person. The ACCU has some amazing personalities and I'm sure we only scratch the surface most of the time.

Each issue of CVu will have someone different. If you would like to share your Desert Island Books please email me: paul.grenyer@gmail.com.

Inspirational (P)articles

Frances Buontempo takes a positive view.

Having recently tended to sink into a sense of hopelessness and despair after swapping software engineering and support war stories with friends, acquaintances and random strangers, I decided I wanted to snap out of it. I fell across an interview [1] with Knuth just in time. This inspired me, and I want to start a series of short, inspiring articles. Please feel free to submit your stories of encouragement and hope.

Knuth talked about many things in this interview, including TeX and how many difficult problems he had to solve which didn't become apparent until he started coding it. He quotes George Forsythe, saying 'People have said you don't understand something until you've taught it in a class. The truth is you don't really understand something until you've taught it to a computer.' He hadn't realised how big the project was, or how long it would take. He acknowledged the impossible task he had given two graduate students to complete as a summer project. Somebody acknowledging programming can be difficult cheered me up. It can be difficult, but that's where some of the fun comes in.

The inspiration came from midway through the interview. Knuth states, 'I checked the other day and found I wrote over 35 programs in January, and 28 or 29 in February. These are small programs, but I have a compulsion. I love to write programs.'

I realised I hadn't written a whole program in a very long time. I have added new features to existing code and fixed bugs, and probably created a few as well. I have experimented with small functions while trying to learn new things. I have talked to people about design and algorithms. I have read books and thought about writing code. But I hadn't written a program. In ages. Since then, I have tried to write one program a week: clearly not as many as Knuth's one or more day, but a start. These are usually small things. For example, I am currently reading a book on machine learning, so I am trying to code some of the algorithms it describes. And it is fun. I love programming. I'd forgotten, but I remember now.

References

[0] Terry Pratchett, can't recall which book(s)

[1] ACM Communications: Aug 2008 'Interview: Donald Knuth. A life's work interrupted'

G'OO'd Behaviour

London Regional Meeting – 19th March 2009.

Steve Love talked about OO design and development to a packed room of twenty or so people, including some new faces. The subject was inspired by a question from a colleague: Are there any recent books/articles on good OO design, since the classic texts tend to be older, weighty tomes, advocating RUP and a plethora of UML diagrams?

As a starting point, Steve suggested four principles: simplicity, roles and responsibilities, equality and the value of values. Simplicity came first, as it underpins all other considerations. Try to avoid complicated ways of doing things. Try to avoid neat tricks. Try to make you code as clear as possible. Next, considering the roles and responsibilities of your objects leads to simplicity, less coupling and more clarity. Avoid having objects or functions with multiple responsibilities. Don't use inheritance inappropriately. Remember the Liskov substitution principles. Consider what makes two objects equal. Do they refer to the same place in memory? Do they have the same state? Finally, disambiguate monomorphic and polymorphic types. If an object is a value type, define it as a value type.

After the presentation, members of the audience threw in some other suggestions. The extent to which the distinction between reference and value types really mattered was considered. The importance of unit tests and the impact of TDD was touched on, but the general feeling was this differed from OO design, though supported the four bywords advocated in the talk. Somehow we ended up discussing singletons, at which point a halt was called to proceedings and we adjourned to a local tavern.

Desert Island Books (continued)

After this major cull, my first musical choice is *Marquee Moon* by Television. Coming from the same New York scene that produced The Ramones, Blondie and Talking Heads, this must have been the least punk-like record ever released by a punk group. A 10 minute long title track, lengthy guitar solos, all very seventies rawk. But the guitars are needle-sharp, and the players the epitome of pale skinniness – when listening to this album, it is hard to believe music can get any more thrilling.

I have to admit though, that the appeal of *Marquee Moon* is purely visceral; it's all about the sound rather than the songs. Despite the singer taking the name Verlaine, the lyrics are hardly poetry. As a contrast, my final choice is Nick Drake's *Fruit Tree*, a collection filled with eloquent songwriting. Now I have to admit that I'm bending my own self-imposed rule here, as this is actually a box-set of Drake's entire recorded output. But his career was brief, so at a stretch I'm claiming it as a single choice. Drake released three albums of brooding, melodic, folk-influenced pop in the late 1960s and early 1970s which went almost entirely unnoticed at the time before his early death in uncertain circumstances. His music has since been rediscovered and become popular to the point where he is almost the archetypal cult artist. But even without the mystique that has grown around him, the remarkable thing is what a unique and fully formed talent he was. Almost every song is a gem, and a delicate, melancholic mood is sustained throughout. It is, perhaps, not the cheeriest choice to take along on a lonely sojourn, but it is among the most beautiful music that I know, and so I can think of no better companion to see me through until my rescuers arrive. They are on their way, aren't they?!?

Next issue: Gail Ollis picks her desert island books.

Bookcase

The latest roundup of book reviews.

If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamourous 'not recommended' rating, you are entitled to another book completely free.

I must thank Blackwells and Computer Bookshop for their continued support in providing us with books.

Jez Higgins (jez@jezuk.co.uk)

Simply Rails 2.0

By Patrick Lenz, published by Sitepoint, 472 pages, ISBN-13: 978-0-98904552-0-5

Reviewed by Simon Sebright

Recommended with reservations.

I read this book rather quickly, which was a good sign; it is written in an easy style and well presented. After some introductions, it takes the form of developing single application, vaguely amusingly called Shovell after the Diggit app.

The book is aimed at novice or someone looking to move to Rails. This is an unfortunate combination, as witnessed by the introduction to OOP.

The introductions have the usual how to get Ruby and Rails installed things, which cover the main OSes adequately. Then there is a low point in the book – an introduction to object-oriented programming, particularly with Ruby. The painful example of Cars and Types of Car is used, which any experienced OO programmer would cringe at.

Anyway, we get stuck into the first skeleton of the Shovell app we create by about page 100. If you have hung on till then, the development of the app, describing Ruby's full stack capabilities is well explained. The features are explained well with a good description of the key new points the code illustrates.

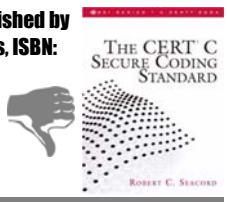
We start off with a basic model and GUI, and then move on to Web 2.0 things, including Ajax and effects, and then on to security issues. I particularly liked the attention to detail on the testing capabilities of Rails. You can run unit tests on the model, functional tests on controllers, and full user-stories replaying the interaction of a user through a complete use case (e.g. log on, add a thing, edit the thing, log out).

At the end, there is a rather cursory chapter on deploying to a production environment. I haven't actually done that yet, but feel it'll be more than the few steps outlined, particularly getting access to the right parts of the system via your hosting provider.

The RESTful system of URLs is well explained, indeed the application is based on a REST model. Unfortunately, I found a lot of the Rails conventions are not actually explained, rather they are simply taken for granted. This is fine, as long as you don't make a mistake in, for example, the plurality of something when creating a model or controller. It pays to take care.

All in all, I enjoyed the book, and found myself referencing it now and again whilst developing my own ideas. However, there is a much more comprehensive text from the Pragmatic Programmers on Ruby on Rails, which I would recommend over this one. Certainly if you are a novice, it could be a good introduction to whet your appetite, but it doesn't contain any kind of reference, which you will need to develop anything of merit.

The CERT C Secure Coding Standard



By Robert C. Seacord, published by Addison Wesley, 720 pages, ISBN: 0321563212

Reviewed by Derek M Jones

Not recommended.

This book follows in a long tradition of providing recommendations to users of a programming language on how to write more reliable software. While the title and contents use the word 'security', many of the recommendations have the more general audience of developers interested in writing reliable software.

The 14 chapters are divided by language and library topic (e.g., preprocessor, floating-point, memory management, signals) and each chapter contains around 10–20 or so recommendations on that topic. Recommendations follow the same format, containing one or more of the following subsections: general discussion, noncompliant code example, compliant solution, exceptions, risk assessment and references. The 612 pages include an extensive index and code examples are the minimum needed to illustrate an issue.

The material assumes that readers have a lot of background knowledge. Each recommendation jumps right in and talks about very specific technical issues often with no background explanation. While some recommendations are labeled as being Microsoft Windows specific most of the material is OS agnostic, although it does make assumptions that best fit a desktop environment (e.g., the `int` type is 32-bits).

The recommendations could be grouped by where they came from. Many recommend against using constructs or relying on behavior that the C Standard specifies as being undefined, implementation defined or unspecified, others recommend doing things that could be categorized as 'good' coding practices aimed at avoiding a source of commonly encountered programmer mistakes (e.g., 'Do not reuse variable names in subscopes' and 'Do not use floating-point variables as loop counters'), while others are platitudes (e.g., 'Understand how arrays work' and 'Take care when creating format strings').

I have been in the source code analysis business, mostly involving C, for almost 20 years and in many places I found I had to read the material several times to understand the point that was being made. The discussion often has little narrative thread, jumping around between topics and introducing topics unrelated to the issue at hand. The wording of some recommendations bears little resemblance to the actual coding issue, e.g., 'Do not apply operators expecting one type to data of an incompatible type' is the recommendation wording used to describe the problems that occur when a pointer is assigned the address of an object having an incompatible type and that pointer is then dereferenced to obtain a value.

Platitudes can be ignored, but every time I encountered one – I stopped counting at 20 – my opinion of the recommendations dropped. But overly generalised recommendations will result

Bookshops

The following bookshops actively support ACCU (offering a post free service to UK members – if you ever have a problem with this, please let me know – I can only act on problems that you tell me about). We hope that you will give preference to them. If a bookshop in your area is willing to display ACCU publicity material or otherwise support ACCU, please let us know so they can be added to the list

- **Holborn Books Ltd** (020 7831 0022)
  www.holbornbooks.co.uk
- **Blackwell's Bookshop**, Oxford (01865 792792)
  blackwells.extra@blackwell.co.uk

in an unnecessary increase in costs if followed, e.g., the recommendation 'Do not simultaneously open the same file multiple times' does not apply if all of the files simultaneously open are for reading.

While it contains lots of recommendations it still manages to miss some of the more commonly encountered ones, not comparing floating point values for equality being the most obvious omission.

The 'Risk Assessment' subsection is a good idea that seems to have been poorly implemented. No information is given on how the likelihood of a recommendation not being followed and the severity of the consequences was calculated; values appear to have been pulled out of the air. From my own measurements of source code I know that some of the likelihoods are incorrect. Severity is very hard to judge, an uninitialise variable may result in a coffee machine occasionally failing to add/omit sugar or firing a missile attached to an aircraft still in its hanger (I know of a case where this happened).

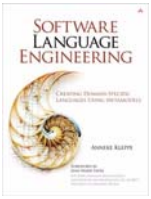
This book is the paper edition of the information available on the web site: https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard

I have bought paper versions of books whose PDFs are freely available on the web. Is it worth paying £38.99 for a paper copy of this web site (which is actively updated)? Unless the paper version contains lots of colour diagrams, 611 pages for £38.99 is a poor deal.

This book contains a significant amount of material that still needs lots of work to make it understandable to the non-expert programmer and to fix the numerous small technical errors.

This book should really have been written to accompany the web site, providing background information and an overview of each of the major topics. As a paper version of the web site it is poor value for money and for material that still needs lots of work.

Software Language Engineering: Creating domain specific languages using metamodels



By Anneke Kleppe, published by Addison Wesley, 240 pages, ISBN: 0321553454

Reviewed by Derek M Jones

This is a Teddy bear book. Are we sitting comfortably? Then I shall begin. A long time ago in a far away place a little girl, perhaps having a second childhood, decided to study terribly hard for a very important qualification. After staying up very late and studying very very hard it was time to write down everything she had learned.

This being a Teddy bear book our heroine did not want to upset people by having original thoughts or talking with messy things like practical details. Recycling pearls of received wisdom found in trusted 40 year old books (Fred Brooks' *The Mythical Man-Month* and that old favorite the *Dragon* book) is so much more comforting. Feeling adventurous our heroine sprinkled some pearls found in Wikipedia, safe in the knowledge that fact-challenged articles could always be edited later.

Language and engineering are awfully long words and perhaps some of Teddy's friends might get lost. Reworking and repeating different chapters and sentences can be so helpful to people who want to stay out of the dark forest.

But a Pee aytch Dee is a terribly important qualification and sometimes magical spells need to be included to impress the men who have to be impressed ('... L is a metamodel those nodes can represent elements that can be materialized to the human senses ...'). Fear not dear reader, the spells' dark bold fonts are easy to see and averting your eyes in their presence will not cause you to get lost.

Time to go to bed.

Effective Java, 2nd Edition

By Joshua Bloch, published by Prentice Hall, 384 pages, ISBN: 0321356683

Reviewed by John Lear

Highly recommended

The first edition of *Effective Java* was published in 2001, this, the second edition, refreshes the original content and adds additional detail to that present in the first edition. As its name suggests, this book follows the same format laid out by Scott Meyers in his *Effective C++* series. The intention of mimicking such a well renowned series sets the bar fairly high. Fortunately, this book does not disappoint.

Split up into 11 main sections, with individual items below this, the book covers everything from general programming tips, through classes, interfaces and generics and finishes up in the more hard-core areas of concurrency and serialisation. The main reason for the update has been the addition of a section on Generics which accounts for approximately one third of the new material. The first edition, published in 2001, could only cover language features up to Java 1.3. The remaining new additions are spread throughout the remaining sections bolstering their content.

I had started this review with the intention of reading the book cover-to-cover, but as is often the case, time pressure have meant that I reverted to dipping in and out of the various sections of the book. It is very easy to do this, with each individual item largely self contained and easily readable. What is also important in a book like this is the quality of the index. This book has a good, detailed index so it is easy to find items that cover a particular issue. What I also like about this book is the fact that it does go into some detail on what the JVM is actually doing where it is appropriate to the issue being covered.

I have really enjoyed reading *Effective Java* and have found the content universally useful when ever I have picked it up be that for general reading or if I have encountered a particular issue in my work. If you are a Java developer you should have this on your bookcase.

Clean Code

By Robert Martin, published by Prentice Hall, 464 pages, ISBN: 0132350884

Reviewed by Seb Rose

Uncle Bob is one of the keynotes at this year's ACCU conference, so I thought I'd have a look at his latest book. It is subtitled 'A Handbook of Agile Software Craftmanship', but like so many books with the 'A' word in them these days, it is much more widely applicable than that.

In many ways it covers similar gound to that covered by Kent Beck's *Implementation Patterns*, but is much more thorough and useful. However, this book also deals exclusively in Java, sometimes becoming too Java-specific for my tastes. The illustrations, provided by Martin's daughter, didn't do it for me either.

The book is divided into three parts (The Theory, Worked Examples, An Index of Smells and Heuristics) and two appendices.

The first section comprises 13 chapters that deal with writing Clean Code – code that's easy to read, easy to test and easy to change. He starts at the beginning, with advice on naming variables, methods, classes. Next is functions – 'Keep them small. How small? Smaller than that?' And so on, through commenting, formatting, data structures, error handling, unit tests, class design and system design.

There is lots of good advice here. You may think some of it is pointless, but then think about some of the worst code you have ever looked at. Now think about some of the worst code you have ever written. You may already have learnt most, or all, of the advice in this book, and you may disagree with some of it, but it is bad advice that makes you think again about your habits.

A few of these chapters are not written by Robert Martin. A number of his colleagues contribute to some of the chapters and not all of these are as useful (or well written) as Bob's. A particular weak point for me was a chapter entitled 'Systems' (contributed by Dr. Kevin Dean Wampler) that delved too deep into certain Java libraries and techniques.

The Worked Examples are interesting and useful, but I don't believe his premise that by working through them you will necessarily absorb the theory. You must apply the principles in practice to fully benefit from the book, but following the Worked Examples is neither necessary nor sufficient.

The chapter that lists the smells and heuristics is an invaluable index and summary of the material

View From The Chair

Jez Higgins
chair@accu.org

Crashing the deadlines as ever, I'm writing this while making lunch and listening to Gardener's Question Time on the radio[1]. I'm not big into gardening myself other than by proxy, indeed my wife is at the allotment as I type, although I am happy to cook up the results. I haven't bothered to retune because, in the same way that I don't really follow *The Archers*, there's a certain comforting familiarity about GQT. Unthreatening amateur gardeners lob up gentle problems to the panel of experts. They, in turn, dispense sage advice, generally after poking good-natured fun at the questioner or one another. How reassuring it must be for gardeners across the nation and, through the miracle of the internet, around the world. Any question answered, any problem solved, any difficulty overcome in two or three minutes of harmless banter. If only programming were as straightforward ...

> We're finding it awkward to work out which branches we need to apply bug fixes to. What does that panel advise?
>
> Now, have you been pruning back hard enough?

<chuckles>

By the time this issue of CVu hits your doormat, 2009's ACCU conference will have come and gone. Despite the doom and gloom of the credit crunch being all around us, Giovanni and his conference committee have put together a top notch programme and registrations have held up extremely well. It should be great. Our AGM also takes place during the conference. It will be my penultimate AGM as Chair, as I will be standing down at next year's. ACCU will, therefore, be in need of a new Chair. A few years ago, and as peculiar as it may sound, I had a vision of myself as Chair during a committee meeting while sitting on floor of Alan Bellingham's front room. Shortly afterwards, Ewan stood down, and my destiny was made manifest. Or something like that anyway. Running ACCU isn't an especially difficult job, particularly with the assistance of the other officers and committee, so if you've recently had a precognition of your future or are merely curious as to what it entails please get in touch.

[1] What's more, a radio running code written by ACCU members

Membership

Mick Brooks
accumembership@accu.org

I've just finished running through the membership statistics for the AGM. They show a small, but disappointing, drop in numbers for the second year running. Last year I believed the drop was related to teething troubles for the new membership system at the major renewal time in August. It seems that wasn't the full story: this year everything has been running smoothly and yet there's still been a drop in membership.

I know many of you do a great job evangelizing for ACCU. I'd love to hear how you get on talking to colleages and friends. Is there an approach you've found that works? Or even that doesn't work? You all presumably stay members because you get value from ACCU. I've found it hard to convincingly make a case for that value. We do have great tangible stuff: the magazines, the mailing lists, the conference, the local groups, but I still think there's much more to ACCU than that. There's definitely something to be said about communicating with people who care deeply about programming and doing it as well as we know how – but I find it hard to put that across without making us sound like a bunch of self-satisfied know-it-alls. Can you do it?

As always, please send any questions or suggestions about membership and renewals to me at accumembership@accu.org.

Advertising

Seb Rose
ads@accu.org

The website advertising continues to be fully subscribed, and we have recently begun taking text links from Cylon (which you may not have noticed). However, we are looking for more journal advertisers, so if you know of anyone who would be looking to benefit from an association with the ACCU, please put them in touch with ads@accu.org

Bookcase (continued)

presented in the first section of the book. I haven't been able to read it in its entirety, but the use I have made of it demonstrated its utility.

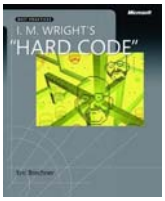
The first appendix deals with in-depth issues arising from concurrency. Concurrency is covered briefly in an earlier chapter, but this appendixdelves deeper into the murky waters.

The second appendix lists an implementation of the open source date library that is used in the worked examples.

I.M. Wright's 'Hard Code'

By Eric Brechner, published by Microsoft Press, 240 pages, ISBN: 0735624351

Reviewed by Seb Rose



Brechner is Director of Development Excellence at Microsoft and has been writing internal blog columns under the I.M. Wright pseudonym since 2001. This book is apparently the complete set of those internal articles, unmodified and arranged into something approaching a coherent structure. Some sidebars have been added to document Microsoft specific terminology, changes in I.M. Wright's opinion and the occasional additional observation.

The book's target audience ranges from junior developer through to management with advice that covers all levels of process and practice. It is divided into 10 chapters that cover much of the pain of working in large software organisations with titles that range from 'Project Mismanagement' via 'Software Design If We Have Time' through to 'Being a Manager, and Yet Not Evil Incarnate'. Each chapter starts with an introduction that sets the context for the articles it contains.

I.M. Wright has an opinionated, undiplomatic voice. He describes failings and fallacies where he sees them and tries, not unsuccessfully, to be controversial and argumentative. I enjoyed both his observations and his suggested tactics.

In the few places that he extolls the virtues of Microsoft as a customer focused organisation I found myself unable to suspend disbelief sufficiently, so I just skipped them – just a few paragraphs. What he did get across, however, was the feeling that Microsoft was at least trying to address some of the problems that such a large organisation is liable to suffer from.

Brechner is clearly a bright guy, and has spent much of the last decade looking at ways to improve processes within, and performance across, the organisation. He has read widely and cites some interesting books, authors and papers (the ommission of a bibliography is a great shame).

The book is well written and not at all dull, with lots of insights for anyone interested in trying to come to terms with working in a large organisation. The pieces are short enough to allow a piecemeal approach to reading the book, which I particularly liked.