the magazine of the accu

www.accu.org

{cvu}

Volume 26 • Issue 1 • March 2014 • £3

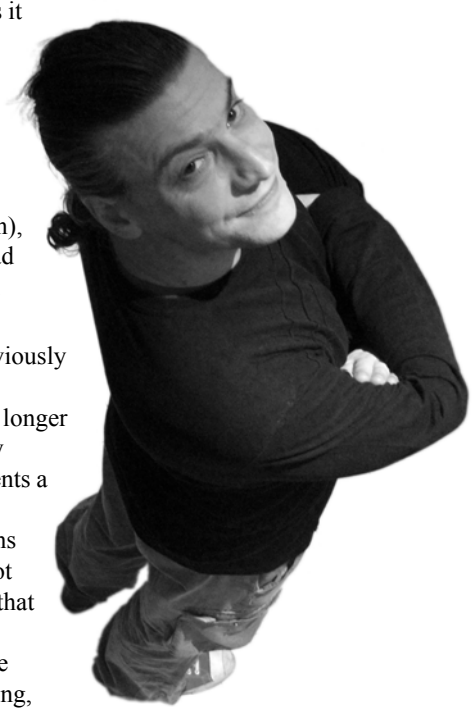
Features

Regulars

{cvu}
Volume 26 Issue 1
March 2014
ISSN 1354-3164
www.accu.org

accu

The Ecumenical Programmer



What is ACCU? What is it about? What does it represent? What's it *for*? When ACCU began, it was all about C and C++ – the latter in particular. It's important to put it in it's *time* to understand why: C++ was undergoing its first round of ISO standardisation. ACCU was an informal bridge between 'users' and 'experts' (although it's impossible to make a clear distinction), and provided a platform for people to write and read about the latest developments in the C++ Standard.

Since those early days, ACCU has changed much. Although the magazines are still being printed (obviously – you're reading one!), the content has broadened immensely. Similarly, the annual Conference is no longer as strongly focussed on the C++ element that really dominated its early days. Today, the ACCU represents a much wider community of software developers working across the industry from embedded systems to large-scale cloud-based web-applications. It's not that C++ or C have become irrelevant to us, rather that other technologies have become more important. Crucially I think, ACCU is about more than just the technologies: it is about the **Practice** of programming, which is so much more than just typing in code. The difference between good and bad coding practice is important to developers and managers alike – and even to users. The most important thing to managers and users is that they shouldn't have to care – they should be able to assume it.

And that's what ACCU is. It is the community around software development that fosters good practice and provides a way for people from lots of different disciplines – whether from different programming languages, different aspects of a team, or different sectors of the industry – to find out about new things, and share their experiences. We all have something to learn from each other, and we all have something to share. Whether you're a beginning programmer coding for fun, an old hand trying to make your first steps on the management ladder, or someone who already manages a team, ACCU is relevant to *everyone* involved in the business of producing software.

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 26.2: 1st April 2014
C Vu 26.3: 1st June 2014

Overload 121: 1st May 2014
Overload 122: 1st July 2014

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

Where Linq Contains a Defect

Glen Fury shares his investigation of a hidden defect.

It's Christmas day. On Boxing Day, new software would be in control of setting the sale prices for my employer's online store. It was the first real run of the application, the application had been in use for almost a week, and was able to change the price of individual products. However, I had never tried to update the prices of all ~3000 products at once. A serious testing oversight.

I decided Christmas night, my belly full, my head spinning just right, I should give it a test run. I was rostered on for support: if it did not work they would call me anyway. I connected to the VPN and setup my machine and simulated the price update for Boxing Day. The result: a **`StackOverflowException`**, crashing the whole application. Merry Christmas.

I found the code causing the problem (see Listing 1) when changing the prices of all the products in our catalogue **`skus`** and **`sites`** contained ~3000 records. I did some quick tests with small alterations to the code. I found the segment **`sku.Contains() && site.Contains()`** was causing the problem. How or why such a function would cause a **`StackOverflowException`** I had no idea. But with not much time before Boxing Day, I tried something quickly and found a fix that worked and deployed it (Listing 2). This is one of the few cases where I thought it was better to ask forgiveness than permission.

The fix worked the application set the prices ready for Boxing Day.

Listing 1

```
string[] skus = skuSitePrices.Select(ssp => ssp.SKU.Identifier).ToArray();
string[] sites = skuSitePrices.Select(ssp2 => ssp2.SiteId.ShortName).ToArray();
IList<ProductSitePrice> productSitePrices =
unitOfWork.ProductSitePriceRepository.FindAll().Where(
psp => skus.Contains(psp.Product.SKU) &&
sites.Contains(psp.Site.SiteName)).ToList();
/*
The FindAll()  returns a DbSet<ProductSitePrice> object.
The application uses EntityFramework4 The Where returns an IQueryable
(note the insertion of ToList())
*/
```

Listing 2

```
string[] skus = skuSitePrices.Select(ssp => ssp.SKU.Identifier).ToArray();
string[] sites = skuSitePrices.Select(ssp2 => ssp2.SiteId.ShortName).ToArray();
IList<ProductSitePrice> productSitePrices =
unitOfWork.ProductSitePriceRepository.FindAll().ToList().Where(
psp => skus.Contains(psp.Product.SKU) &&
sites.Contains(psp.Site.SiteName)).ToList();
/*
(note the insertion of ToList())
*/
```

Listing 3

```
ISet<string> skus = new HashSet<string>(skuSitePrices
.Select(ssp => ssp.SKU.Identifier);
ISet<string> sites = new HashSet<string>(skuSitePrices
.Select(ssp2 => ssp2.SiteId.ShortName));
IList<ProductSitePrice> productSitePrices =
unitOfWork.ProductSitePriceRepository.FindAll().Where(
psp => skus.Contains(psp.Product.SKU) &&
sites.Contains(psp.Site.SiteName)).ToList();
```

Further investigations

January when our dev lead came back from his Christmas break I reported the problem. He was incredulous, so I recreated the problem by turning back the clock to Boxing Day and running the application. Thankfully it did throw a **`StackOverflowException`**.

The source of the code was now looked into closely. I was the original author, and can say that the code was written as an attempted optimisation: to grab from the repository only the products required. On review we found what I had written was inefficient. I had chosen to use arrays, this allowed for duplicate string values to exist in the variables **`skus`** and **`sites`**. So an adjustment was made we altered the code to use **`HashSet`**s (Listing 3). That way the code might achieve what it was designed to, and remove the **`StackOverflowException`**.

This was successful, the code ran without crashing with a **`StackOverflowException`**. Although the execution was much slower than we expected, taking about a minute to run.

We theorised that it may have just masked the exception by reducing the amount of stack it required. To test this hypothesis we doubled the **`HashSet`**'s size (Listing 4).

The result of the testing was a **`StackOverflowException`**. So the **`HashSet`** did not fix the problem. So we tried changing the point of the execution to retrieve the data by removing the **`.ToList()`** (see Listing 5).

This solution did not work either, as it also threw a **`StackOverflowException`**. At this point I began to speculate on the value of reducing the set of products pulled from the database. There were only ~3000 products anyway. In this instance, a full site sale, nearly all the products were updated. Why not forgo the optimisation especially as it seemed to expose a defect in Linq, and Entity Framework 4.

At this point we started tracking how long the function took to run under different solutions. I also tried the code without the **`Where`** function (Listing 6).

This worked. However, I wanted to try one more option (Listing 7) – Hot Fix with Hashsets.

This worked without throwing a **`StackOverflowException`**. Under the instruction and with the help of the Dev Lead, we tested the code solutions above in 32-bit and 64-bit builds, and with 3000, and 6000 products so that we might understand the defect better. The results are shown in Table 1.

GLEN FURY

Glen's dad taught him how to program for fun when he was a teenager. He's been programming professionally since 2006 mostly in e-commerce, and for fun again since mid 2013. He can be contacted at furg0998@gmail.com

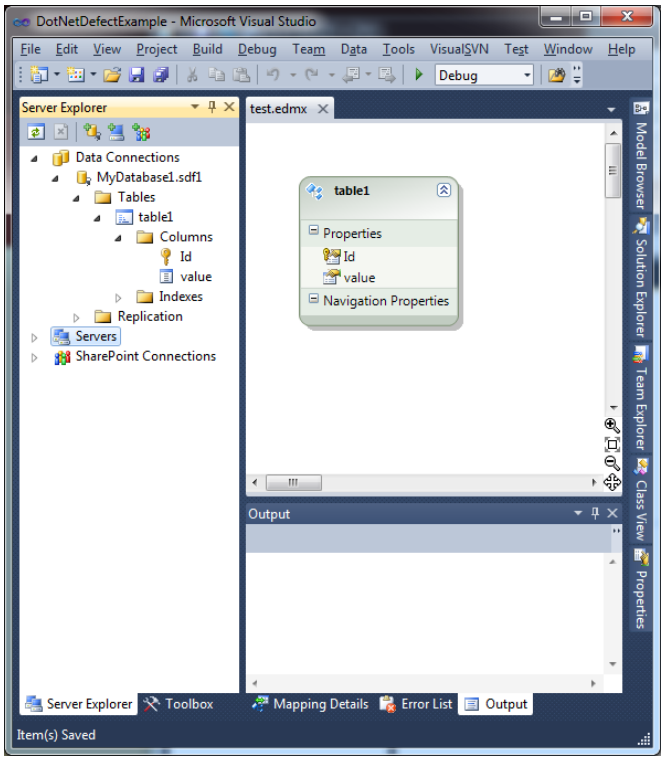
Experimenting

A large set or array run against a database using the `Contains` method has an upper limit, and performance seemed sluggish. Why did running the same Linq statement run so much quicker on a list?

I came up with the hypothesis, that the Array or Set was being used with an Entity Framework dbset was unpacked by a recursive function, and thus causing a `StackOverflowException`. What else causes a `StackOverflowException`? So I set about experimenting to see if I could prove my hypothesis.

Firstly I wanted to see if the issue was peculiar to the application, or it could be reproduced with a basic example. So I started a new console project, created a simple edmx and created a database, but left the database empty. I thought perhaps an empty database might not have the same problem, and that adding data to the database might skew any performance results (Figure 1).

Figure 1



Then I went about trying to reproduce the original problem, and see if I could find an approximate sweet spot where the `StackOverflowException` would occur.

I simply tried to reproduce the defect with the code in Listing 8.

After a couple of tests, where I changed the value of the `ArraySize` I found the `StackOverflowException` happening occurred regularly at 12000, but not at 11500.

I then applied my hotfix to see if it prevented the `StackOverflowException`, and it did. I had successfully replicated the issue.

Listing 4

```
ISet<string> skus = new HashSet<string>(skuSitePrices
.Select(ssp => ssp.SKU.Identifier);
  List<string> _skus = skus.ToList();
  foreach(string sku in _skus)
{
  skus.Add(sku + "_____");
}
ISet<string> sites = new HashSet<string>(skuSitePrices
.Select(ssp2 => ssp2.SiteId.ShortName));
       List<string> _sites = sites.ToList();
  foreach(string site in _sites )
{
  sites.Add(site + "_____");
}
IList<ProductSitePrice> productSitePrices =
unitOfWork.ProductSitePriceRepository.FindAll().Where(
psp => skus.Contains(psp.Product.SKU) &&
sites.Contains(psp.Site.SiteName)).ToList();
```

Listing 5

```
string[] skus = skuSitePrices.Select(ssp =>
ssp.SKU.Identifier).ToArray();
string[] sites = skuSitePrices.Select(ssp2 =>
ssp2.SiteId.ShortName).ToArray();
IQueryable<ProductSitePrice> productSitePrices =
unitOfWork.ProductSitePriceRepository.FindAll().Where(
psp => skus.Contains(psp.Product.SKU) &&
sites.Contains(psp.Site.SiteName));
  ...
  foreach(ProductSitePrice psp in productSitePrices) {
    ...
    /* update the prices */
  }
```

Listing 6

```
IList<ProductSitePrice> productSitePrices =
unitOfWork.ProductSitePriceRepository.FindAll()
.ToList();
```

The `Contains(t1Item.value)` method when ran on a `List` object it performed as expected, but when being run on a database set it did not.

I speculated that as the `where` statement must be translated into SQL at some point when run against a database. So I needed to find out how it was translated to SQL. I expected it would be translated into a statement like

```
SELECT * FROM table1
WHERE value IN ('1', '2', '3', '4' … '5000');
```

as that is how I would write the SQL statement.

I looked at what the argument I was passing to the `Where` method: `t1Item => searchList.Contains(t1Item.value)`. It is an `Expression<Func<table1,bool>>` and the Body of the Expression is a `MethodCallExpression`. The method is `Contains`, and the two parameters are the `t1Item.value` and the other the members of the

Table1

Code Option	32-bit 3000 products	32-bit 6000 products	64-bit 3000 products	64-bit 6000 products
Listing 1: Original	Failed StackOverflow Exception	Failed StackOverflow Exception	Failed StackOverflow Exception	Failed StackOverflow Exception
Listing 2: Hotfix	Success in ~2,000ms	Success in ~5,000ms	Success in ~2,000ms	Success in ~5,000ms
Listing 3: HashSets	Success in ~60,000ms	Success in ~220,000ms	Success in ~65,000ms	Failed StackOverflow Exception
Listing 4: HashSets Doubled	Success in ~230,000ms	Failed StackOverflow Exception	Failed StackOverflow Exception	Failed StackOverflow Exception
Listing 5: Late Execution	Failed StackOverflow Exception	Failed StackOverflow Exception	Failed StackOverflow Exception	Failed StackOverflow Exception
Listing 6: No Filtering	Success in ~2,900ms	Success in ~13,000ms	Success in ~2,800ms	Success in ~8,800ms
Listing 7: Hotfix with Hashset	Success in ~1,600ms	Success in ~4,500ms	Success in ~1,800ms	Success in ~4,500ms

searchList array. This didn't help me decipher what the SQL would look like, as the **MethodCallExpression** still needed to be translated to SQL.

So I reduced the **ArraySize** to 10 and wrote a Linq to SQL statement, and used the **ToTraceString()** method to find what SQL was generated.

```
string result = (from t1 in tc.table1
where searchResults.Contains(t1.value) select t1)
as ObjectQuery<table1>).ToTraceString();
```

Listing 7

```
ISet<string> skus = new HashSet<string>(skuSitePrices
.Select(ssp => ssp.SKU.Identifier);
ISet<string> sites = new HashSet<string>(skuSitePrices
.Select(ssp2 => ssp2.SiteId.ShortName));

IList<ProductSitePrice> productSitePrices =
unitOfWork.ProductSitePriceRepository.FindAll().ToList().Where(
psp => skus.Contains(psp.Product.SKU) &&
sites.Contains(psp.Site.SiteName)).ToList();
```

Listing 8

```
public static int ArraySize = 12000;
static void Main(string[] args)
{
  string[] searchList = new string[ArraySize];
  for (int i = 0; i < ArraySize; i++)
  {
    searchList[i] = i.ToString();
  }
  using (testContainer tc = new testContainer())
  {
    List<table1> result =
    tc.table1.Where(t1Item =>
    searchList.Contains(t1Item.value)).ToList();
  }
}
```

Listing 9

```
public static Expression<Func<table1, bool>>
  GenerateContainsExpression(string[] array)
{
  ParameterExpression paramTable1 =
    Expression.Parameter(typeof(table1),"t1");
  MemberExpression memberExpression =
    LambdaExpression.PropertyOrField(paramTable1,
                                     "value");
  BinaryExpression mainExpression = null;
  for (int i = 0; i < array.Length; i++)
  {
    ConstantExpression constantExpression =
      Expression.Constant(array[i]);
    BinaryExpression expression =
      Expression.Equal(memberExpression,
                       constantExpression);
    if (null == mainExpression)
    {
      mainExpression = expression;
    }
    else {
      mainExpression =
        Expression.OrElse(mainExpression,
                          expression);
    }
  }
  return Expression.Lambda<Func<table1, bool>>
    (mainExpression, new ParameterExpression[]
  { paramTable1 });
}
```

The result was:

```
SELECT
  [Extent1].[Id] AS [Id],
  [Extent1].[value] AS [value]
FROM [table1] AS [Extent1]
WHERE (N'0' = [Extent1].[value]) OR
      (N'1' = [Extent1].[value]) OR
      (N'2' = [Extent1].[value]) OR
      (N'3' = [Extent1].[value]) OR
      (N'4' = [Extent1].[value]) OR
      (N'5' = [Extent1].[value]) OR
      (N'6' = [Extent1].[value]) OR
      (N'7' = [Extent1].[value]) OR
      (N'8' = [Extent1].[value]) OR
      (N'9' = [Extent1].[value])
```

The **Contains** was transformed into a number of **OR** statements, this supported part of my hypothesis, the members of the array were being cycled through one by one. However, it did not prove that there was a recursive function responsible for the translation.

I thought that I should try to do the same, turn the **Contains** into a number of **OR** statements, perhaps I could write a function that did so without causing a **StackOverflowException**. So I wrote the code in Listing 9 to receive my **SearchList** and return an **Expression** of **OR** statements. I then changed my **Where** statement to the following:

```
Expression<Func<table1, bool>> filter =
GenerateContainsExpression(searchList);
List<table1> result =
tc.table1.Where(filter).ToList();
```

Once I confirmed it worked on a small number of items I decided to run it against the original to see which would out perform the other. The results rounded to the nearest 50 milliseconds are shown in Table 2.

My function was outperformed, it caused a **StackOverflowException** to occur when there was only 5500 items in the list. Why? I looked at what I was generating. The expression, is in fact an expression tree, with branches of **OrElse** Binary Expressions. The **StackOverflowException** was not caused by a recursive function generating the expression but rather reading the expression itself used too much stack. If that was the case, why was my function not performing as well? When I looked at the code, it became obvious. My expression tree was very lopsided.

So I rewrote my function to create a less lopsided expression tree (see Listing 10).

I tested this new function, and compared the results to the original. My results were as in Table 3 and Figure 2.

So my new function performed as well as the original. I had found the cause of the problem. A limitation of binary expressions.

This limitation does not seem so bad on its own. To me it makes sense, who would under normal circumstances write an expression tree of that size. But there is the problem. Using **searchList.Contains**, I never expected it to be transformed into a huge expression tree that would be unable to execute. After all it is a very simple function it takes very little time to write, further if I were writing the output in SQL I would have used an **IN** statement.

So there is a problem here allowing a **Contains** in the **Where** method when it is not capable of handling arrays of reasonable size. Entity framework by its nature converting expressions to SQL cannot handle any type of expression. For example you can't compare non entity objects, or custom properties of your entities. If you do try this, Entity Framework will throw a **NotSupportedException**.

Perhaps for a **Contains** it should also throw a **NotSupportedException**, and leave it to the developer to find the best way to generate the expression. It was only through writing a code to

Listing 10

```
public static Expression<Func<table1, bool
GenerateContainsExpression(string[] array)
{
  ParameterExpression paramTable1 =
    Expression.Parameter(typeof(table1),"t1");
  MemberExpression memberExpression =
    LambdaExpression.PropertyOrField(paramTable1,
                                     "value");
  List<BinaryExpression> expressions =
    new List<BinaryExpression>();
  for (int i = 0; i < array.Length; i++)
  {
    ConstantExpression constantExpression =
      Expression.Constant(array[i]);
    expressions.Add(Expression.Equal
      (memberExpression, constantExpression));
  }
  while (expressions.Count > 1)
  {
    List<BinaryExpression> newList =
      new List<BinaryExpression>();
    for (int i = 0; i < expressions.Count;
         i += 2)
    {
      if (i + 1 >= expressions.Count)
      {
        // must be an odd number of expressions
        newList.Add(expressions[i]);
      }
      else
      {
        newList.Add(Expression.OrElse
        (expressions[i], expressions[i + 1]));
      }
    }
  expressions = newList;
  }
  return Expression.Lambda<Func<table1,
    bool>>(expressions[0],
  new ParameterExpression[] { paramTable1 });
}
```

generate an expression of a number of `OR`s that I was able to see the difficulty and problems with applying this to a large array.

The other solution would be to change how a `Contains` is translated into SQL, if it was translated into an SQL `IN` I think would be able to outperform the current implementation. I expect that too would have its limitations.

A further solution would be to create a new type of `BinaryExpression` that could take a list of binary exceptions and a comparison type, eg `OR`, `AND`. This would allow for a much flatter expression tree and `StackOverflowException`s would not be caused. ■

Figure 2

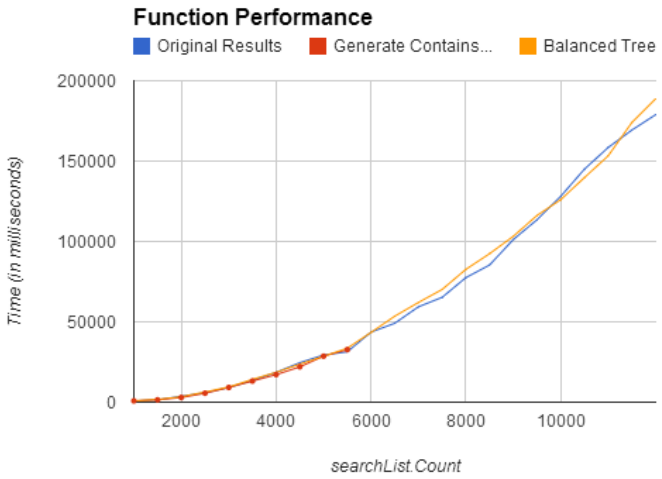


Table 2

Items in searchList	Original (milliseconds)	Generate Contains Expression (milliseconds)
500	750	800
1000	1500	1500
1500	3650	2900
2000	5900	5550
2500	9150	9150
3000	13550	13150
3500	18550	17150
4000	24600	22000
4500	29350	28650
5000	31350	32700
5500	43600	StackOverflow
6000	49100	
6500	59350	
7000	65250	
7500	77500	
8000	85350	
8500	101200	
9000	113500	
9500	128100	
10000	145000	
10500	158450	
11000	169300	
11500	178850	
12000	StackOverflow	

Table 3

Items in searchList	Original (ms)	Generate Contains Expression (ms)	Generate Contains Expression Improved Tree (ms)
500	750	800	850
1000	1500	1500	1700
1500	3650	2900	3400
2000	5900	5550	6350
2500	9150	9150	9450
3000	13550	13150	14200
3500	18550	17150	18600
4000	24600	22000	23700
4500	29350	28650	28550
5000	31350	32700	33600
5500	43600	StackOverflow	43400
6000	49100		53500
6500	59350		62050
7000	65250		70150
7500	77500		82650
8000	85350		92400
8500	101200		103200
9000	113500		116200
9500	128100		126000
10000	145000		139750
10500	158450		153250
11000	169300		174000
11500	178850		188750
12000	StackOverflow		StackOverflow

Software Archaeology

Chris Oldwood digs up some ancient remains.

Don't ever take a fence down
until you know why it was put up
~ Robert Frost

Changing software is hard. It's especially hard when you're new to the team or codebase and don't know the history of the people or the application. Making any change is not just as simple as adding, replacing or removing lines of code. There are also the higher-order aspects to consider, such as whether it fits in stylistically, idiomatically and within the existing design principles. In a really old codebase these can be particularly hard to pin down as they will have undoubtedly changed over time. For example, new code written today under C++ 11 is going to be quite different to code written years ago under C++ 98 and that will need to be factored in when trying to understand what's going on.

I discovered just how hard it was to test the component in isolation

Small-scale archaeology

Putting to one side the stylistic and idiomatic aspects which may well be at the whim of the author, the design can be a lot harder to understand. Many years ago I worked on an old C++ codebase for a big finance institution. My role was purely technical and I was initially tasked with fixing various memory leaks and deadlocks within its highly threaded services. Whilst tracking down one leak I came across some code that looked something like this:

```
ContextPtr getContext(long contextId)
{
  static ContextPtr cachedContext;
  if (cachedContext.get() &&
      cachedContext->Id() == contextId)
      return cachedContext;
  cachedContext = Context::Load(contextId);
  new ContextPtr(cachedContext);
  return cachedContext;
}
```

The spurious extra copy of the smart-pointer was downright weird. I could assume that it was just a cut-and-paste error or dead code, but it seemed too random for that, especially in such a short function. So I decided to consult the version control system to see what it could tell me about its past life.

This function appeared to have a somewhat chequered history. Although it had now been stable for a couple of years its birth was a little more tortuous. The author had originally started without the spurious smart-pointer copy. They then started adding random try/catch blocks in different places, presumably because the code was crashing under mysterious circumstances. Finally the try/catch blocks were gone and there appeared to be a couple of attempts to mess with the smart-pointer value and reference count before settling on the final implementation. But what was going on?

The missing piece of the puzzle is what the object being pointed to contained – a COM object, and a remote one at that. Putting this knowledge together with the static local variable, which we know will be destroyed after `main()` exits, and it all starts to make sense. The author was having lifetime issues because COM was uninitialised before the C++ based context object was destroyed. This caused the DCOM remote proxy to throw an exception during invocation of the destructor during process shutdown. This explains why the `try`/`catch` blocks were presumably having no effect – they were notionally in the wrong place.

Ultimately what the author really needed was a weak-pointer. The full-blown smart-pointer based single item cache was keeping the object alive far longer than necessary. The cached reference was a useful optimisation but the lifetime was deterministically controlled elsewhere. Once I'd worked all this out I switched from the home-brew smart-pointer type to one in Boost and the original problem (the memory leak – a huge one) was fixed.

All this was 'derived' from the version control history and the author hadn't written any check-in comments either which might have saved me time. But there was also some other interesting metadata about the check-ins that I found interesting too – each commit appeared to be followed by a formal build (the file with the build number changed) and was hours apart. This suggested to me that the changes were never tested locally first. After all, why bother to check-in the intermediate steps if they didn't work? This was years before Git was fashionable and there were no automated tests in sight. It appears as if the changes were tested by deploying into the shared development environment and waiting to see what happened.

When I came to test my own changes I discovered just how hard it was to test the component in isolation. In particular the large, monolithic service that formed the heart of the system was not amenable to localised testing at all without some serious refactoring. I concluded that whilst the developer was clearly keen to try and resolve the issue the environment and development process did not seem to be supporting him in making that happen easily.

Tribal knowledge

Sometimes we're lucky and the person who wrote the code is still around so we can try and shortcut some of the digging around. If they're still working on the project you have the opportunity to transfer some of the more implicit design knowledge that seeps away over time. More recently I came across a change to a very simple C# extension method I had written for the `String` class. This was my original version:

```
public static bool IsEmpty(this string value)
{
  return (value.Length() == 0);
}
```

Which had been changed to this:

```
public static bool IsEmpty(this string value)
{
  return String.IsNullOrEmpty(value);
}
```

For those non-C# readers there is a subtle difference in behaviour that happens when the value is a null reference. In my original implementation a `NullReferenceException` will be thrown, whereas in the revised version it will be silently ignored. Looking back in the source code repo I studied the associated changes and realised that this change was made to fix a problem with null string references coming in through the web API

CHRIS OLDWOOD

Chris is a freelance developer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros; these days it's C++ and C#. He also commentates on the Godmanchester duck race. Contact him at gort@cix.co.uk or@chrisoldwood

I always approach any codebase with the assumption that the author is a smarter programmer than me and knows more about the problem domain too

framework in certain scenarios. I proposed that the fix was in the wrong place as it changed the semantics of the extension method [1]. After a brief discussion we refactored the code, added a couple of unit tests to document the extension method behaviour and got on with our lives. Sometimes it's easy to forget how even a simple, 1-line method can hide such detail.

Unearthing patterns

Having the author around doesn't always help though. There are programmers you'll meet who are not quite so amenable to such discussions and will instead get easily offended if they suspect you're questioning their judgment.

As you might deduce from the Robert Frost quote at the very beginning I always approach any codebase with the assumption that the author is a smarter programmer than me and knows more about the problem domain too. At least, until I know for definite that isn't the case. This means when I see code that isn't obviously simple [2] I err on the side of caution and assume that I might be about to learn something new. This is almost always true because even if I don't learn something new about the language, libraries or problem domain I will probably learn something about the author or environment instead.

One time I came across the following declaration in a C++ codebase:

```
std::vector<std::string*> hostnames;
```

My instinct clearly told me this was wrong as RAII almost always rules, but being new to the project and having the author just feet away I politely quizzed them on it. The response was a very confrontational "*Are you reviewing my code?*" After trying my best to explain my good intentions I escaped with the knowledge I sought – the code was wrong. More importantly I came away knowing a lot more about the author's character.

Once you see some code where a programmer has made what could be considered 'a fundamental mistake', you immediately begin to question where else they may have made it. The version control system can tell you what other commits someone's made and with a sprinkling of command-line magic it's often possible to see where they've been and what they've touched. In this instance I restricted myself to just the most critical areas of the code but it paid dividends as I chalked up a couple more memory leaks.

Although somewhat jaded by this experience I decided it was still my duty to point out to an ex-project member, who only worked down the hall, that 19 of the memory leaks I'd just unearthed were due to him forgetting to mark the base class destructor virtual even though they were managed by a smart-pointer. Fortunately he was far more grateful for the feedback.

Blame

One of the most useful tools in modern source control systems is Blame. Whilst its name might suggest it be used for nefarious purposes, such as starting a witch-hunt, it would be far better thought of as 'Excavate'. Starting with a revision it will show you who last touched each line of code and what commit it came from. From there you can trace back through the past changes looking at the evolution of the file. Sadly the one thing it doesn't show is where code has been deleted which can be a useful lead some times.

Revision graph

The other key tool in the VCS arsenal is the revision graph. This tool shows you the birds-eye view of a file's history – what commits have taken place, their check-in comments, what branches changes have been made on, how and when they were merged and what labels or tags have been applied.

If you have a bug and you want to know what versions of your product the bug appears in this is almost certainly your starting point.

TICOSA

I've mostly only been concerned with small-scale software archaeology as it's usually directly related to a change I've needed to make. Occasionally I've done some larger scale spelunking, such as rating contributors by their percentage of empty check-in comments, but nothing much more than that.

In January of this year I attended The (First) International Conference on Software Archaeology [3] at the Museum of London. There were a number of other ACCU members present and we spent the day finding out a bit more about how software archaeology might be used in other ways. Given its infancy, the message was fairly clear that whilst analysing the history of a codebase can be interesting, we're still not sure exactly what the value is. Nonetheless it was a useful day and hopefully in subsequent years we'll get to find out how it can be used more effectively. ■

References

[1] http://chrisoldwood.blogspot.co.uk/2013/09/extension-methods-should-behave-like.html
[2] http://en.wikipedia.org/wiki/Tony_Hoare#Quotations
[3] http://ticosa.org/

Write for us!

C Vu and Overload rely on article contributions from members. That's you! Without articles there are no magazines. We need articles at all levels of software development experience; you don't have to write about rocket science or brain surgery.

What do you have to contribute?

- What are you doing right now?
- What technology are you using?
- What did you just explain to someone?
- What techniques and idioms are you using?

For further information, contact the editors: cvu@accu.org or overload@accu.org

Wallowing in Filth

Pete Goodliffe sinks into some terrible code.

As a dog returns to its own vomit,
so fools repeat their folly
Psalms 26:11

We've all encountered it: *quicksand code*. You wade into it unawares, and pretty soon you get thank sinking feeling. The code is dense, not malleable, and resists any effort made to move it. The more effort you put in, the deeper you get sucked in. It's the man-trap of the digital age.

How does the effective programmer approach code that is, to be polite, *not so great*? What are our strategies for *coping with crap*?

Don't panic, don your sand-proof trousers, and we'll wade in...

Smell the signs

Some code is great. Like fine art, or well-crafted poetry. It has discernible structure, recognisable cadences, well-paced meter, a coherence and beauty that make it enjoyable to read and a pleasure to work with.

But, sadly, that is not always the case.

Some code is messy and unstructured: a slalom of `goto`s that hide any semblance of algorithm. Some is hard to read: with poor layout and shabby naming. Some code is cursed with an unnecessarily rigid structure: nasty coupling and poor cohesion. Some code has poor factoring: entwining UI code with low-level logic. Some code is riddled with duplication: making the project larger and more complex than it need be, whilst harbouring the exact same bug many times over. Some code commits 'OO abuse': inheriting for all the wrong reasons, tightly associating parts of code that have no real need to be bound. Some code sits like a pernicious cuckoo in the nest: C# written in the style of JavaScript.

Some code has even more insidious badness: brittle behaviour where a change in one place causes a seemingly unconnected module to fail. The very definition of *code chaos theory*. Some code suffers from poor threading behaviour: employing inappropriate thread primitives or exercising a total lack of understanding of the safe concurrent use of resources. This problem can be very hard to spot, reproduce, and diagnose, as it manifests so intermittently.

(I know I shouldn't moan, but sometimes I swear that programmers shouldn't be allowed to type the word *thread* without first obtaining a licence to wield such a dangerous weapon.)

> Be prepared to encounter bad code. Fill your toolbox with sharp tools to deal with it.

To work effectively with alien code, you need to able to quickly spot these kinds of problem, and understand how to respond.

Wading into the cesspit

The first step is to take a realistic survey of the coding crime scene. You arrive at the shores of new code. What *are* you wading into?

The code may have been given to you with a pre-attached stigma. No one wants to touch it because they know it's foul. Some quicksand code you discover yourself when you feel yourself sinking.

It's all too easy to pick up new code and dismiss it because it's not written in the style you'd prefer. Is it really dire work? Is it truly *quicksand code*, or is it merely unfamiliar? Don't make snap judgements about the code, or the authors who produced it, until you've spent some time investigating.

Take care not to make this personal.

Understand that few people set out to write shoddy code. Some filthy code was simply written by a less capable programmer. Or by a capable programmer on a bad day. Once you learn a new technique or pick up a team's preferred idiom, code that seemed perfectly fine a month ago is an embarrassing mess now and requires refactoring.

You can't expect any code, even your own, to be perfect.

> Silence the feeling of revulsion when you encounter 'bad' code. Instead, look for ways to practically improve it.

The survey says...

We've already looked at techniques for navigating a new codebase in 'Navigating a Route' [1].

As you build a mental model of a new piece of code, you can begin to gauge it's quality using benchmarks like:

- Are the external APIs clean and sensible?
- Are the types used well-chosen, and well named?
- Is the code layout neat and consistent? (Whilst this is certainly not a guarantee of underlying code quality, I do find that inconsistent, messy, code tends also to be poorly structured and hard to work with. Programmers who aim for high-quality malleable code also tend to care about clean, clear presentation. But don't base your judgement on presentation alone.)
- Is the structure of co-operating objects simple and clear to see? Or does control flow unpredictably around the codebase?
- Can you easily determine where to find the code that produces a certain effect?

It can be hard to perform this initial survey. Maybe you don't know the technology involved, or the problem domain. You may not be familiar with coding style.

Consider employing *software archaeology* in your survey: mine your revision control system logs for hints about the quality. Determine: how old is this code? How old is it in relation to the entire project? How many people have worked on it over time? When was it last changed? Are any recent contributors still working on the project; can you ask them for information about the code? How many bugs have been found and fixed in this area? Many bug fixes centred here indicates that the code is poor.

Working in the sandpit

You've identified quicksand code, and you are now on the alert. You need a sound strategy to work with it.

What is the appropriate plan of attack?

- Should you repair the bad code?
- Should you perform the minimal adjustment necessary to solve your current problem, and then run away?

PETE GOODLIFFE

Pete Goodliffe is a programmer 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 or @petegoodliffe

- Should you cut out the necrotic code and replace it with new, better work?

Gaze into your crystal ball. Often the right answer will be informed by your future plans. How long will you be working with this section of code? Knowing that you will be pitching camp and working here for a while influences the amount of investment you'll put in. Don't attempt a sweeping re-write if you haven't the time.

Also, consider how frequently this code has been modified up to now. Financial advisors will tell you that "past performance is not an indicator of future results". But often it is. Invest your time wisely. This code might be unpleasant, but if it has been working adequately for years without tinkering, it is probably inappropriate to 'tidy it up' now, especially if you're unlikely to need to make many more changes in the future.

> Pick your battles. Consider carefully whether you should invest time and effort in 'tidying up' bad code. It may be pragmatic to leave alone it right now.

If you determine that it is not appropriate to embark on a massive code re-work right now, that doesn't mean you are necessarily left to drift in a sea of sewage. You can wrestle back some control of the code by cleaning progressively...

Cleaning up messes

Whether you're digging in for the long haul, or just making a simple fix-and-run, heed Robert Martin's advice and follow the 'Boy Scout rule': "Always leave the camp ground cleaner than you found it." It might not be appropriate to make a sweeping improvement today, but that doesn't mean you can't make the world a slightly less awful place.

> Follow the *Boy Scout Rule*. Whenever you touch some code leave it *better* than you found it.

This can be a simple change: address inconstant layout, correct a misleading variable name, simplify a complex conditional clause, split a long method into smaller, well-named sub-functions.

If you regularly visit a section of code, and each time leave it slightly better than it was, then before long you'll wind up with something that might be classified as *good*.

Making adjustments

The single most important advice when working with messy code is this:

> Make code changes slowly, and carefully. Make one change at a time.

This is so important that I'd like you to stop, go back, and read it again.

There are many practical ways to follow this advice. Specifically:

- Do not change code layout whilst adjusting functionality. Tidy up the layout, if you must. Then commit your code. Only *then* make functional changes. (However, it's preferable to preserve the existing layout unless it's so bad that it gets in the way.)
- Do everything you can to ensure that your 'tidying' preserves existing behaviour. Review and inspect your changes. Get extra sets of eyeballs on it. This is the prime directive of *refactoring*: the well-known set of techniques for improving code structure. This goal only be reached effectively if the code is wrapped in a sound set of unit tests. It is likely that messy code will not have any tests in place; so consider whether you should first write some tests to capture important code behaviour.
- Adjust the APIs that wrap the code without directly modifying the internal logic. Correct naming, parameter types and ordering; generally introduce consistency. Perhaps introduce a new outer interface; the interface you wish that that code had. Implement it in terms of the existing API. Then at a later date you can re-work the code behind that interface.

The curious case of the container code

There was a container class. It was central to our project. Internally, it was foul. The API stank, too. The original coder had worked hard to wreak code mischief. The bugs in it were hidden by the already confusing behaviour. Indeed, the confusing behaviour was a bug itself.

One of our programmers, a highly skilled developer, tried to refactor and repair this container. He kept the external interface intact, and improved many internal qualities: the correctness of the methods, the buggy object lifetime behaviour, performance, and code elegance.

He took out nasty, ugly, simplistic, stupid code and replaced it with the polar opposite. But in his effort to maintain the old API this new version was internally far too contrived, more like a science project than useful code. It was hard to work with. Although it succinctly expressed the old (bizarre) behaviour, there was no room for extension.

We struggled to work with this new version, too. It had been a wasted effort.

Later on, another developer simplified the way we used the container: removing the weirder requirements, therefore simplifying the API. This was a relatively simple adjustment to the project. Inside the container, we removed swathes of code. The class was simpler, smaller, and easier to verify.

Sometimes you have to think laterally to see the right improvement.

Have courage in your ability to change the code. You have a safety net: source control. If you make a mistake, you can always go back in time and try again. It's probably not wasted effort, as you will have learnt about the code and its adaptability in doing so.

Sometimes it is worth boldly ripping out code, in order to replace it. Badly maintained code that has seen no tidying or refactoring can be too painful and hard to correct piecemeal. There is an inherent danger in replacing code wholesale, though: the unreadable mess of special cases *might* be like that for a reason. Each bodge and code hack encodes an important piece of functionality that has been uncovered through bitter experience. Ignore these subtle behaviours at your peril.

An excellent book that deals with making appropriate changes in quicksand code is Micheal Feather's *Working Effectively with Legacy Code* [2] In it, he describes sound techniques to introduce seams into the code; places where you can introduce test points and most safely introduce sanity.

Bad code? Bad programmers?

Yes, it's frustrating to be slowed down by bad code. The effective programmer does not only deal well with the bad code, but also with the people that wrote it. It is not helpful to apportion blame for code problems. People don't tend to purposefully write drivel.

> There is no need to apportion blame for 'bad' code.

Perhaps the original author didn't understand the utility of code refactoring, or see a way to express the logic neatly. It's just as likely there are other similar things you do not yet understand. Perhaps they felt under pressure to work fast and had to cut corners (believing the lie that it helps you get there faster; it rarely does).

But of course, you know better.

If you can: *enjoy* the chance to tidy. It can be very rewarding to bring structure and sanity to a mess. Rather than see it as a tedious exercise, look at it as a chance to introduce higher quality.

Treat it as a lesson. Learn. How will you avoid repeating these same coding mistakes yourself?

The Soundtrack to Code 2: Going Classical

Silas S. Brown gives us his take on 'Music for Coding'.

This is a response to Adam Tomhill's article 'The Soundtrack to Code' in *C Vu* January 2014. My suggestions for moments when music is required to aid concentration and/or reduce distractions will always be classical music, but readers who are not experts in that field might have difficulty choosing which pieces to try.

As a preliminary experiment to see if this can work for you, I highly recommend downloading the Open Goldberg Variations from www.opengoldbergvariations.org – this was a KickStarter-funded project to produce a high-quality public-domain recording of Bach's Goldberg variations, played on the piano by German-Japanese pianist Kimiko Ishizaka. This will give you about an hour and a half of gentle piano music which sounds more 'natural' than any synthesizer and may well help you concentrate. Try the Goldbergs – go on, try it (it won't cost anything to download) – if it doesn't work for you then feel free to skip the rest of this article, but give it a chance.

If you can't get enough of Bach played that well on the piano, then after the Goldbergs I suggest looking for a recording of the 48 Preludes and Fugues, otherwise known as The Well-Tempered Clavier, for example the one played by Daniel Barenboim. Warner Classics sells a 5 CD boxed set, giving you nearly 5 hours of music (if you load them all onto the computer you don't have to change the CDs). It's important to use a good recording that brings out the feel of the music rather than just playing notes.

Also worth looking out for is Bach's Brandenburg Concertos (instrumental, 2 hours); get a good period-instruments recording if possible. You might also want to try the nine Beethoven symphonies on period instruments (Gardiner, or the Hanover Band); if the name 'Beethoven' makes you think of dreary modern performances by the likes of von Karajan then you might be pleasantly surprised by the sound of the 'period performance' (historically-informed performance) movement. Basically, as musical instruments have been developed over the years, the hardware the music was originally written for had somewhat different characteristics to what we have now, so to bring out its original beauty you need to be aware of these differences and perhaps use a good emulation. (Think of trying to play classic games on modern hardware that gets the speed all wrong for example.)

Another 'boxed set' of symphonies you might like to try is Schubert's (9 symphonies, 4.5 hours). You might be able to borrow this and others from a public library before committing to buying it. (Some libraries let you listen to music in the library on headphones without charge; take your laptop and figure out the Wi-Fi.)

Other names to look out for: Sammartini (concerti and sonatas), Scarlatti, Colpron, Pamela Thorby (a very good recorder player who digs up all kinds of interesting early music), Vivaldi (and no, I don't just mean the Four Seasons: try his concertos), Telemann (many instrumental works), Guerrieri (sonatas), and perhaps Heinichen. All of this is 'early music' originally written for the soundtracks of royal courts and such; if it was good enough to accompany the writing of their master-plans or whatever, then it might also do for your coding.

Note that I'm deliberately steering clear of the so-called 'popular classics', because I'm assuming that most people have written those off as too boring, or have heard them too often in shops, etc. There are many more interesting and unusual instrumental works which can make good 'thinking music' and also enjoyable listening, so don't write off 'classical' until you've tried a good variety of it. ■

SILAS S. BROWN

Silas S. Brown is a partially-sighted Computer Science post-doc in Cambridge who currently works in part-time assistant tuition. He has been an ACCU member since 1994 and can be contacted at ssb22@cam.ac.uk

Wallowing in Filth (continued)

Check your attitude as you make improvements. You might think that you know better than the original author. But do you always know better?

I've seen this story play out many times: a junior programmer 'fixed' a more experienced programmer's work, with the commit message 'refactored the code to look neater'. The code indeed looked neater. But he had removed important functionality. The original author later reverted the change with the commit message: 'refactored code back to working'. ■

Questions

1. Why does code frequently get so messy?
2. How can we avoid this from happening in first place? Can we?
3. What are the advantages of making layout changes separately from code changes?
4. How many times have you been confronted with distasteful code? How often was this code *really* dire, rather than 'not to your taste'?

References

[1] 'Navigating a Route' Pete Goodliffe. In: *CVu* Volume 24 Issue 6, January 2013. ISSN: 1354-3164

[2] *Working Effectively with Legacy Code*. Michael Feathers. ISBN: 978-0131177055

Developer Freedom

Chris Oldwood muses on the liberties we should and should not enjoy.

All employees are equal, but some employees are more equal than others
~ George Orwell (mostly)

When it comes to the subject of access to the Internet developers are quite clearly far more equal than any other sort of employee in the business. Or at least some think so, but are we?

Over the last decade I've found myself working at some big corporations – the kind of places where IT is a part of the business, but not the actual business itself (despite what you might choose to believe about its importance today). As a consequence there is almost a Mexican standoff between the security team, whose purpose is to keep the company safe, and the developers/testers/support staff whose job is to provide and maintain solutions to solve business problems. To perform this function effectively invariably requires accessing some content and/or downloading additional tools that the company does not already provide for. So what's the big deal?

malicious content has evolved from being the result of misguided individuals with something to prove, to being a business

Fear, uncertainty and doubt

The landscape has changed dramatically over the last 20 years where malicious content has evolved from being the result of misguided individuals with something to prove, to being a business – if you consider organised crime a business that is. At least, that's if you believe what the security industry tells us. Throw in the recent revelations about government spying and I don't think it's hard to see why the paranoia levels in the security departments of these big corporations are at Spinal Tap [1] levels.

Take my own personal web site as an example of where the level of corporate trust is almost certainly very low. I have always made the source code available alongside every tool I've ever published as a courtesy to anyone who might be interested. But let's face it, how many developers download the source code, sift through it to make sure there aren't any exploits, then build it and finally see if it's going to be useful? Virtually none, I'd wager. In fact I'd question whether the licence agreement even gets an airing.

No, what we do is see if the web site looks legitimate, i.e. it's not just a random IP address for an FTP site somewhere on the planet, and if we think it looks trustworthy we'll go ahead and try it. In the case of the really popular sites, like NuGet, I bet we don't even give security a second thought; after all, if big companies like Microsoft are posting content on there it has to be totally legit, right? From a security team's perspective, seeing how some of us behave, I'd suggest that free, 3rd Party components and frameworks are like dancing pigs [2] for developers.

OK, I get that executable content can be really dangerous in the same way that granting my normal account admin rights on the production system is dangerous; I want to be protected from my own stupidity. What I definitely don't get though is the apparent danger caused by non-executable content, like blogs. Are 'The Powers That Be' afraid we'll somehow become subverted by poisonous articles that will generate an uprising to overthrow the management? Or are they just afraid we'll waste time by looking at the football scores which will undoubtedly lead to even more wasted time as we argue over the relative merits of a 4-4-2 formation? If the worry is that we might try and sell the source code for profit, then I think they have an over-inflated notion of 'quality' as plenty of the enterprise code I've seen would be worth less than the USB stick used to traffic it!

Talking of admin rights, why is so hard to obtain those on my local machine? Luckily plenty of software copes these days without needing local admin rights. In fact I'm writing this on Word 2002 using an LUA-style [3] account under Windows XP. Even modern versions of Visual Studio play nicely, but there are still times when elevation is required by my job – unless testing, debugging and deployment has been removed from my job description. Once again, what are they afraid of? Surely the worst damage I can really do is screw up my own machine? Or perhaps they're afraid I'll install DOOM and flood the network with IPX/SPX traffic? Is it my problem the company can't partition its network effectively to isolate critical services from those under test? Installing DOOM is not an accident, but designing and building a service where it's all too easy to attach the development and production instances together by accident is equally negligent.

Enterprise developers

The only other answer I can come up with is that I'm 'the wrong sort of developer'. Is the vast majority of software development in the enterprise actually the writing of macros for Excel and doing customisation of 3rd party products? If the development of custom services (native or managed) is in the minority I can see how we 'builders' might appear to be so overly demanding compared to the 'tweakers'. Perhaps there is an assumption too that the modern technique of unit testing helps us to eliminate all those nasty external dependencies and so reduces the need to do any sort of system level testing on our own machines, right? In fact isn't that why they have a QA department?

So far in this article I've pretty much failed to be even vaguely objective, and that really was my goal when writing it. We all know what the *status quo* is; the question is how to overcome it. What can we do to try and convince those in control that to do our jobs effectively we need the reins to be loosened so that we can access more of the internet than our peers? Whilst unfettered internet access for all is possibly the desired end state, with a focus on educating employees to act responsibly, I'm not convinced that's a realistic expectation for an enterprise in the short term to medium term.

Static content

When it comes to non-executable content I don't believe we need to have any more rights than someone in, say, the Accounts or Marketing department. In a large organisation with open-plan offices it should be as culturally unacceptable to view inappropriate content as it would be to

CHRIS OLDWOOD

Chris is a freelance developer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros; these days it's C++ and C#. He also commentates on the Godmanchester duck race. Contact him at gort@cix.co.uk or@chrisoldwood

stick Page 3 pin-ups around your desk. In an agile working environment the demands of constant communication make it virtually impossible to do anything other than your job as you'll be collaborating regularly. And that's before you take into account the effects of pair programming for keeping you honest.

What I suspect the non-developer employees don't realise is quite how much we rely on the internet to do our job. Whilst there are the obvious vendor support sites for the core products we use, there are also the big self-help sites like Stack Overflow. But even allowing access to these is only part of the equation because often the simple answer is just not enough and the salient details are in some blog post that is then linked to. Like it or not blogs are the modern knowledge base for programmers so categorising them as 'personal pages' along with Twitter and Facebook is to completely miss the point. When your job is also your hobby, which it is for many in our profession, then the meaning of 'personal' no longer distinguishes it from 'professional'.

the demands of constant communication make it virtually impossible to do anything other than your job

When they introduced a draconian content filter at a major financial institution I had recently started at, I decided to seek out the team responsible for the change and try to describe how much pain they were causing and to see if they couldn't loosen it somewhat. After a few days of quizzing everyone I knew I managed to track down a chap in the security department and met with him to put forward my case. After showing some examples of blogs that were clearly relevant to 'the business of programming' he accepted that the filter was too coarse. However he played the 'but it's group policy' card to explain why content categorised by the 3rd party content filter as 'social' was forbidden. Exceptions, he said, could be made, but he also intimated that the process for getting content checked and accepted was not a priority.

Later, back at my desk I noticed that any content could also be given a secondary category. Knowing that the primary category of 'Computers/Internet' was allowed I went back to the security team with a proposal. If I could convince the 3rd party content filter vendor to re-categorise blocked blog content from simply 'personal pages' to the more specific 'personal pages and computers/internet' would they allow it? They said 'yes' and added that re-categorisation by the vendor was much quicker than their process.

OK, so this is far from the perfect outcome, but it did feel like I had at least managed to make some progress as I opened access to a number of popular blogs. More importantly I had found out who to contact and got to discuss the issue with them. In the end I decided the only way to instigate any further change would be to show how an outage was a direct result of an inability to do my job properly so that there was a monetary value to the loss. Unsurprisingly the right opportunity never arose because I just needed to get on with my job and I'd got used to using my phone to view blocked content instead. I briefly looked into trying to claim my phone bill back as an expense, but that just created far more pain for me (being a contractor) than for them due to the ridiculous paperwork involved.

Executable content

Access to static information is only one cornerstone of our jobs; another is tools. This includes both entire programs, such as the classic UNIX command line utilities, and libraries which we consume directly within our own applications. Whilst it might be an interesting personal exercise to write an XML or JSON parser, that's not really the best way for us to be spending our customer's money. Component-level reuse is mandatory if we are to create our own applications as efficiently as possible, unless there is an especially good reason to build them ourselves (e.g. legal). The same goes for tools – we shouldn't be writing our own compiler or web browser either.

This rather thorny issue is probably where a large part of the problem lies. Executable content is mostly opaque, except for scripts or where you're building it from source code, and that means it's hard to trust by default. Anyone who has ever been involved in a virus clean-up operation knows how time-consuming it can be. Most organisations now run virus scanners by default and although they do interfere with compilation and development duties, the impact is minimal unless it's set to scan everything (which sadly does happen). One option to reduce their impact is to get a company-wide policy agreeing that certain folders (e.g. C:\Work) or certain processes (e.g. DEVENV.EXE) are excluded.

Aside from a virus attack, allowing developers and support staff to being able to run arbitrary programs, especially ones they don't truly understand, also comes with a risk of consuming other resources, such as network bandwidth. One morning I arrived at work faced with investigating why our clients couldn't access our services. It turned out we'd used up our entire month's bandwidth allowance overnight. This it transpired was down to someone not understanding how BitTorrent works. Aside from the direct financial cost of paying for more bandwidth, there is an indirect cost too in a loss of confidence by their clients.

There is also the subject of licence agreements. Most blog posts don't come with a multi-page licence agreement for you to consume before reading on – programs and libraries do, however. Ensuring that your staff are only using correctly licensed software is hard. Apart from the obvious common collections, like the UNIX toolset, there are many smaller, more specialised tools that we need only temporarily to solve an immediate problem. However we might not even know if it's the right tool for the job without first trying it out – the classic problem of chickens and eggs. If choice is going to be limited, then there must also be an understanding that we'll have to resort to using the wrong tool for the job and that comes with a price.

The ironic thing is one of the reasons we choose to use third party tools and libraries in the first place is because we know writing security conscious stuff is hard and most of us are not qualified to do that reliably ourselves. By increasing security around their employees they have inadvertently reduced their ability to produce secure applications.

Black-listing dangerous content is never going to workable, so the only recourse will be white-listing. The question is whether it's possible to put together a white-list with enough low-risk software that would strike a balance between providing enough of the most common tools we need, without leaving anything obvious out. Whilst the perfect tool for a one-off problem might be a ground-breaking new programming language, I'd wager the problem is also solvable, albeit with a little more effort, in one of the more common, general-purpose languages such as C++ or Python. Specialised tools definitely have their place, but they also have a cost, and getting access to them is just one.

Local admin rights

Disallowing local admin rights on a machine for a normal user is a good defence-in-depth measure; it helps the user to protect themselves from their own mistakes. But as we've just established, developers need these same rights because they often have to evaluate tools. If they're expected to handle deployment or do some form of local end-to-end testing they may need to install, configure and debug their applications whilst running as services. The alternative is essentially remote debugging, which is a painful experience at the best of times. It also takes a test environment out of action and creates a single point of contention which is the whole reason we have our own machines in the first place.

we know writing security conscious stuff is hard and most of us are not qualified to do that reliably ourselves

Once again the problem is no doubt one of trust. The theory must be that if you allow someone the right to install software, any software, they will clearly use it to abuse their position. Whilst we've seen that it's possible to create bigger problems by being granted such powers, this kind of problem only goes to highlight a lack of partitioning within the organisations infrastructure. Production services must be isolated from development services and there should be some kind of airlock required to bridge them; in fact this problem is probably one of the best adverts for cloud-based computing.

if you can't trust your developers and support staff to be given admin rights to their own machines how on earth are you going to trust them with the keys to crown jewels

Ultimately though, if you can't trust your developers and support staff to be given admin rights to their own machines how on earth are you going to trust them with the keys to crown jewels? I've tried to have a grown-up, responsible conversation in the past to help establish trust by promoting a policy where my day-to-day account should not also be used for support. I was told that, whilst it sounded like a good idea in theory, they didn't have enough licences for their 3rd party account management tool to allow it. I then suggested that the cost of accidental failure would probably dwarf that, but my comment was not appreciated.

If I really had to give up this right my (Windows) development machine would need to come pre-configured with a bare minimum of: Visual Studio, the VCS client, a decent Notepad replacement and Gnu on Windows (or equivalent). Of course I'd be the first one out the door the moment the contract came up for renewal.

Who's our champion?

They say every problem in Computer Science can be solved using an extra level of indirection, can that work here? One option would be to stop doing in-house development altogether and just outsource it all. That way we'd be working for a company whose bread-and-butter is IT and so they stand a better chance of understanding our needs. Sadly the rise of Agile means that close collaboration with our customer is called for and that puts us straight back into the client's offices again, but this time with even less influence.

The other candidate for overseeing our well being in a big corporation is the Enterprise Architecture team. These people are allegedly the gate keepers of the company's IT strategy. Their role, as I've always understood it, is to look after the big picture and I can't think of a bigger IT picture than providing the basic tools that every architect, developer, tester and administrator needs to do their job. Sadly it's an exclusive club concerned only with 'design'; what programmers apparently do is merely 'an implementation detail'. Where is the equivalent department for us? There is no 'Enterprise Implementation' team that I've heard of.

The closet thing I have ever seen to something like this was called The Technology Council. Their role was to try and maintain a level of consistency across the various tool chains that the in-house projects needed so that the skill sets of both its developers and support staff were more portable across its applications. It also tackled the licensing issue and potential impedance mismatch problems between applications and operations. However, it didn't seem to include internet access as one of its mandates.

The only other approach I've thought of would be to take a leaf out of the *eXtreme Programming* manual and get someone from the security team to sit with us in a pair programming kind of way. Then they might at least see what we face and the compromises we have to make.

Whatever solutions we come up with starts with us finding the right person to talk to and that's often the first hurdle which we stumble on straight away. Blocked content is usually presented in a 'big brother' fashion with stern words to make you feel intimated. Nowhere on the page does it give you details of the person or team you should consult if you feel access to the content would be beneficial to getting your job done and is therefore in fact in the businesses own interest in granting you permission.

Summary

As a freelance programmer I live and work at the bottom of the corporate food chain. As such I know I can't begin to imagine what it must be like to try and manage even a small company, let alone a large corporation with thousands of workers, each with different roles and abilities. I know it's not personal and that I'm only being tarred with the same brush as the rest of the workers because it's easier to create a homogeneous environment.

But surely there must be a way forward; a way for me to do my job without even having to contemplate doing things that either put me out of pocket, risk violating the terms of my contract or just make me look incompetent because everything takes longer than it should. ■

References

[1] http://en.wikipedia.org/wiki/Up_to_eleven
[2] http://en.wikipedia.org/wiki/Dancing_pigs
[3] http://en.wikipedia.org/wiki/Principle_of_least_privilege

Staying in Touch: Performative Negotiation

Vsevolod Vlaskine joins the dots between three practices to reduce technical cost.

Three expensive things

Coupling, staying out of touch, and speculation, none of them being intrinsically evil (e.g, a quick concept-proof prototype may be highly coupled), all of them have a very high price tag. However, since their price is the time wasted in the (near) future, just like with credit cards, many of us tend to turn a blind eye to the inevitable pain of payback.

Coupling: rule of the second use case

It is one of the most expensive things in software development.

Writing decoupled – i.e. having no unnecessary dependencies – code does not take much longer, but it requires the additional effort of the constant concentration not only on the problem domain, but on the semantic quality of one's libraries or applications.

Since in its early days, the success of a project often is measured in terms of speedy 'modelling' of the problem domain rather than sound codebase, the engineers tend to go slack on the latter.

However, the coupled code scales very poorly: the initially upbeat speed of development inevitably gets reduced exponentially almost to a standstill: the coupled code allows less and less change and natural growth, but only hacks; it contains more and more murky corners; the team gets stuck in permanent fire-fighting and reverse engineering of each other's code. Then the management intervenes with measures rarely targeting the right problem, things somewhat improve, and the project keeps limping from crisis to crisis.

A trivial, but common example: protocol packet parsing gets coupled with the transport protocol, for example in a single function. Both may have bugs. In case of a failure, we cannot tell straight away what caused the bug: parsing or packet acquisition. To get a rough gauge: if there are 3 bugs in the parsing and 5 in the transport, the bugs will interfere with each other and thus the number of failures will multiply, as will the effort of their localization. Due to this multiplication effect, the complexity of a system will grow exponentially, as well as the maintenance time, unless the system is designed in a decoupled way where each element can be tested, debugged, and refactored independently. The price of coupling (e.g. in terms of effort) is exponential.

Thus, relentless decoupling. The engineers often stop at the point, when the components are 'simple enough', rather than going for the rigorous quest for the minimum semantically cohesive vocabulary orthogonal in itself and with other problem domains.

Any amount of coupling, even in the obvious things, introduces a multiplicative component in the price tag of a class or utility. Thus, points of coupling piling on top of each other add an exponential component to the price tag of the whole system.

A while ago, I wrote a library, in which one class had a boolean member. `True` was the natural default value for it, but I set it to `false` by default, since it was more convenient for the system I was working on. The library was a success and found a broad usage across a number of projects. However, as its usage spread it soon exposed the annoying dent: most of the time, the user had to construct an instance of the class and then set that member to `true` by hand. If she forgot to do that, it would produce strange behaviour, and every time it would take fifteen minutes to realise what was going on. I coupled two design considerations, dozens of applications used the class, thus changing the default to `true` potentially could break lots of things, and we had to live with the inconvenience.

The most dramatic decoupling of the code happens with the second use case. The second use case introduces new force, semantic tension in the class usage, which most of the time calls for more decoupling in it.

Fitting the code to a second use case is different from generalization. Generalization may follow from the second use case, but it is not the same. Generalization is finding abstract features that are common: say, generalizing a collection of countable objects as a templated vector. The second use case emphasizes not what is in common, but what is different.

As a simple example, assume, we have a map of city roads, which we load from a file and then perform some search on it. Should our roadmap class load from file on construction? Or have a method `load()`? If we have a second use case where the roadmap is a result of cutting a map region, i.e. loading does not semantically belong to it, which suggests that for a better decoupling `load()` perhaps should be a standalone deserialization function.

In the absence of anything else, the testing pretty much becomes such a second use case, since the automated test suite imposes quite a different usage onto the software artefact under test: for the practical usage of the class the most mainstream scenarios are most important, whereas the test with a good coverage focuses much more on marginal cases. Testing also forces us to decouple the cumbersome construction of a class from its use, avoiding involved mocking, etc: the class or library should offer itself for testing. Whenever it is hard to tell from the test code what is being tested, or the test does not demonstrate artefact usage, or mocking in it is not brought to the minimum, it indicates lack of design in the artefact and introduces coupling between the artefact and its test. Whenever an engineer, if asked why his code is not well-tested, complains that 'that functionality is inherently hard to test', you almost certainly will find a lot of coupling in his code.

The second use makes the well-decoupled code shine. But when the second use is resolved by even more coupling, it locks the coupled code: once two semantically different usages pile on top of it, it is much more difficult to refactor.

Staying out of touch

Efficient communication is one of the main premises in agile – and one of the most violated ones. Whenever there is a lack of direct communication between stakeholders – no direct touch, the engineer most certainly will make wrong design and coding decisions, since he is forced to do too much too early, assume, guess, and pre-empt, rather than wait for the right information. There is nothing terribly bad about it, it just is very expensive: any wrong decision based on lack of information means branching in the development tree. Any branching introduces exponential component into the development time. This puts a quantitative estimate on the agile communication tenet: whenever it is broken, the price rockets exponentially.

Staying out of touch has been plaguing the absolute majority of the projects I have seen or heard of: the project leader would not have time to communicate with his team, the team would not be in contact with the

VSEVOLOD VLASKINE

Vsevolod Vlaskine has over 15 years of programming experience. Currently, he leads a software team at the Australian Centre for Field Robotics, University of Sydney. He can be contacted at vsevolod.vlaskine@gmail.com

clients, the engineers would have no faintest idea about each other's work, etc.

'Staying in touch' expresses an immediate connection when someone keeps a tactile contact with something else of different nature. It is not meddling or intrusion, but a contact by touch only – meaning that there should not be any separating distance either. Such a notion of touch is not fuzzy or subjective. It has to be stringent to work.

For example, in physics it is causal relations. The things that don't produce any effects are out of touch with our world. In mathematics it is exactness. The mathematics is resting not so much on its philosophical foundations that always have been disputed and notoriously wobbly (e.g. due the existential status of mathematical objects or their truth value). The exactitude of mathematical touch is not about the truth of its foundations, but about having no gaps in each step of the proof. For arts and crafts the method of refined touch is quite literal (see e.g. [1]). The famous saying of William Morris, "You can't have art without resistance in the material" means that craft exists only at the point touch.

This might be another reason why writing software is so often compared to arts, craft, which, I think, is not a romantic metaphor, but a reference to a method. As for the commonality between software and mathematics, it lies to a large extent in the rigour of leaving no gaps – but not in the same way. I will talk about the software way below.

Speculation is cheap

Not at all, since it is a special case of staying out of touch – but one that seems to be abused most often.

A typical situation: a design discussion that drags forever because of all the 'maybe' and 'should'. Each 'maybe' or 'if' is a point of branching in the discussion introducing an exponential component into the time spent. Speculative reasoning may look like a cheap way of logically exploring all the possibilities before doing actual work, but when the reasoning tries to reach too far, the branching discussions start seriously eating into the development time.

In the same manner, sometimes during the planning people suggest, say: let us try those three libraries and then choose, after all it takes just two days for each. Trying this and that: sometimes it has to be done, but in general, when you decide to try several cheap things, it means that your development tree forks, introducing an exponential component into the development time.

Discussions consume expensive resources. 15 minutes of 4 engineers talking equals to 1 man-hour that could have been spent on writing code. Rather than saying discussions are useless, I simply say: they are too expensive to be wasted.

The other common part of speculative meetings is pursuing social or political purposes, or asserting someone's ego. I feel it is extremely damaging for the software process, but maybe those things are important for social bonding, establishing common values, or selling the ideas. The analysis of these aspects requires a different method and I would not go there. In my experience, it is important to spot and separate them from the technical purpose-driven conversations.

'Technical' not necessarily means 'design'. It can be about planning, setting up collaborative environment, a strategy of interviewing job candidates – anything with a clear outcome or problem to solve. And if things are fuzzy, then we need to make an effort to distinguish between two sources of vagueness: lack of decisive facts to inform the discussion on one hand, and 'politics' on the other.

Mostly, we cannot obtain the missing decisive facts just by talking, but only by making pragmatic steps of doing something. After a couple of 'maybe' or 'if', it makes sense to pause and instead figure out what would reduce uncertainty before getting back to the discussion: instead of speculating on both possible outcomes, find what action would rule out one of the branches. E.g. we could try to build a concept-proof prototype, run a decisive test, contact the stakeholder who knows, etc.

Moreover, although it may sound counterintuitive, but even if people argue about a specific design decision, as healthy a discussion as it may seem, in all my experience the best course of action is to stop the dispute, write down the assumptions of both parties, and come up with an experiment or proof of concept that would resolve the argument. This is similar to the validated learning in Eric Ries' *Lean Startup* [2]. Most of the time, it is not about one's logical argumentation to convince, but about forces to balance (as in pattern languages): unlike the logical arguments that need to be refuted, the forces do not need to be cancelled, but just balanced. Therefore, a good decisive experiment not so much negates one argument and validates the other, but rather prototypes a design balancing the forces. (In such a game of proofs, it is easy to get carried away, though. Therefore, at any moment I keep asking: 'What is the capability we are trying to achieve?')

As for the political part, my pragmatic step is branching it into another meeting, whenever possible. For example, it is common during a design talk that people dig their heels on a general topic like language choice, parallelism, software process, etc. As fun as it may be, it is rarely productive. If my position permits, I always suggest as early as I can: 'Yes, it is a big important thing, which deserves a separate conversation.' I urge the most vehement party to later choose time, agenda, and invite everyone. Interestingly, in my experience, those follow-up meetings never happen.

Performative negotiation

Whenever I spot coupling, staying out of touch, or speculation, it induces in me the gut-wrenching feeling probably similar to the one in a computer scientist, when he is told to quickly offer a practical solution to an NP-complete problem, because this is what they are: exponential, stressful, and intractable – in the case of software process, for no good reason most of the time.

Semantic coupling commonly happens, when an engineer works in solitary long enough. It is not about time: once he stops articulating what he is doing he has been alone long enough. The quiet concentration time is central to our profession, but it only exacerbates the peril of the over-fitting of the code to a specific use case, which is almost a definition of coupling. To get a second use case or second pair of eyes the programmer has to stop doing and talk instead. Frequent and structured code reviews for one are not so much for catching bugs, but for fixing the expressive aspect of software, making it more meaningful, i.e. usable and intuitive in more contexts. Thus, stop doing and talk.

On the other, a purposeful software discussion should transform a set of practical inputs into a list of concrete actions, which would produce decisive results and pose problems for the next conversation. Thus, stop talking and do.

Brought to the extreme, any action is accompanied with a talk, or rather expressed in it. I call this style of work performative negotiation: any discussion or negotiation is about the meaning of a concrete action in a given context. We articulate any action as we perform it.

The pair programming in XP is exactly an instance of such an extreme. The Scrum stand-up meetings are another example: the talk articulates the progress between yesterday's and today's action.

In other engineering disciplines, they speak about the construction of a bridge or an engine. In software engineering, the communication is an immediate extension of the code. The programming artefacts comprising our vocabularies: libraries, databases, patterns, etc. In a design discussion, we map those vocabularies into the user stories, which become the expressive part of the next step of implementation. The two sides of performative negotiation correspond, or rather are extension of, the two sides the code [3].

Performative negotiation in software engineering is synonymous to staying in touch. The touch between two actions or pragmatic steps is articulated in an expressive vocabulary (in Brandom's words "what needs to be said to be able to do something" [4]); the communication is structured by removing the forking 'if' or fuzzy 'maybe' by a decisive practical step: 'what needs to be done to be able to say something'. [ibid.]

locate your Pi is less clear: their virtual systems might be able to swap out your app when it's inactive, but the Pi runs 24/7 no matter what. A full treatment of power impact is beyond the scope of this article, as it involves where and how the power is generated, how much cooling is needed in that town, and even how they make the magnets when replacing turbines; it might be worth comparing providers on this – Google seems more transparent than Amazon, and GreenQloud in Iceland is worth a look – but I can't see a reason to feel bad about using *any* data centre to replace a home connection. At any rate, the two providers I looked at that are currently offering free small quotas for experimentation – AppEngine and OpenShift – both involve virtualisation.

Google AppEngine and RedHat Openshift are both called Platform as a Service (PaaS) providers: they provide a 'platform' of pre-installed tools (including Python and other programming languages) and APIs which you use without having root access to the virtual machine. This contrasts with Infrastructure as a Service (IaaS) like Amazon's AWS cluster which gives you root access on your own virtual machine which you have to set up yourself. (OpenShift are currently using Amazon AWS as their back-end, and might have some kind of deal, since AWS licenses Red Hat, but I don't have any insider knowledge about cross-licensing deals.) AppEngine and OpenShift's current business models are to allow free use of a small but very reasonable quota for hobbyists and bootstrap startups, in the hope that some of these will go on to have larger requirements that are chargeable. AWS also allows a free small start, but it is time-limited. The other services might also be time-limited but they haven't told us yet. Cloud services can change on the whim of the companies that run them (as was demonstrated with the controversial retirement of Google Latitude *et al*), so my advice is use it while it's there but don't rely on it.

Obviously you wouldn't want to upload any code that you really wouldn't want the cloud company to see, but most of them seem professional enough (after all their reputations are on the line); a larger danger is over-reliance on a specific company's API, so try to avoid that: policies and limits could change at any time, and you never know when you might want to bring the server back in house.

AppEngine

AppEngine was perhaps the simplest environment to set up and also seems easier to scale quickly. I made my Python script [4] use Tornado in multithreaded WSGI mode, which means Google is free to start and stop as many instances as their algorithms and quotas see fit. They give you a wildcard subdomain of appspot.com, or you can use your own domain but for the latter you do need complete control of an entire domain (a simple subdomain of a Dynamic DNS service will not do for AppEngine, although it does currently work on OpenShift).

For AppEngine, I needed to create a file called `app.yaml` with contents like Listing 1.

I placed this file, my script and a symlink to Tornado into a directory (some Tornado versions also need you to create empty 'placeholder' versions of `fcntl.py` and `ssl.py`), and then used the AppEngine SDK's downloadable '`appcfg.py`' tool to push it to the server. It works reasonably well, apart from little nags like inability to access Google's search results (looks like somebody in their wisdom decided that Google search results will not be available to Google AppEngine, so I filed a bug report, which doesn't seem to have been looked at, and went off and used, dare I mention it with Google as my platform provider, a non-Google search engine).

OpenShift

Compared with AppEngine, I found OpenShift a bit more complex to set up and perhaps less efficient (a virtual server instance, which OpenShift calls a 'gear', has to run at all times, rather than WSGI instances being started and stopped on-demand), but it does have some advantages. For one thing, unlike with AppEngine, OpenShift servers may run binaries that you've written in C and compiled yourself (they're not limited to only the supplied interpreted languages). To compile C or C++ (or whatever) you will need to install a RedHat-compatible Linux distribution such as CentOS, which I did using Vagrant and VirtualBox [5] so I didn't have to change my main Debian-based setup; although it's possible to run the compiler on the server they give you, this is best done for small programs only, as it can easily run out of resources; anyway it's usually quicker to compile locally and upload the binary.

Listing 1

```
application: large-print-websites
version: 1
runtime: python27
api_version: 1
threadsafe: true
handlers:
- url: .*
  script: wrapper.myApp
```

Also, OpenShift currently has no trouble using Dynamic DNS hostnames as long as you set OpenShift not to scale your application (unless your Dynamic DNS provider supports CNAME records). OpenShift doesn't seem to 'do' wildcard domains though.

To use OpenShift you need to install a utility called RHC (Red Hat Cloud?) and type (for example) **`rhc setup`** and **`rhc app create myApp python-2.7`**. Then use **`rhc git-clone myApp`** and you have a Git repository. To add Tornado, put it into the **`install_requires`** section of `setup.py`, and run it from app.py. To push all this to the server and start it, simply **`git commit -a`** and **`git push`**.

If you don't want OpenShift to restart your server when pushing, create a file called `.openshift/markers/hot_deploy` and it won't restart, but remember to delete this file before pushing something that you do want to restart, or alternatively use **`rhc app restart`** after pushing, which might lead to slightly less downtime than a normal deploy but I haven't measured this properly.

If you're worried about taking up space, you can delete your git history like this:

```
git update-ref -d refs/heads/master
git commit -m "removed git history to save space"
git push --force
```

and then do **`rhc ssh`** and enter these commands:

```
cd git/*.git
git reflog expire --all --expire-unreachable=now
git gc --aggressive --prune=now
```

but that does have the disadvantage of throwing away your git history, so it probably shouldn't be done unless you're really low on space (or have committed some very large files you really shouldn't have committed).

OpenShift can also run cron jobs (see their documentation for where to put it), although if writing a Python job that calls out to the shell, SHELL might not be something you recognise unless you set it. Mail sent using the default 'mail' command might disappear, due to other SMTP servers rejecting the machines on the EC2 cluster, so you might need to establish a tunnelled connection to your own SMTP server to send mail. ■

References

[1] 'Web Annotation with Modified-Yarowsky and Other Algorithms' *Overload* issue 112 (December 2012) p.4 (PDF p.5)

[2] www.raspberrypi.org

[3] Silent MIDI file (52-byte silent ringtone) http://people.ds.cam.ac.uk/ssb22/compos/noise.html#silent

[4] Web Adjuster http://people.ds.cam.ac.uk/ssb22/adjuster/

[5] www.vagrantup.com (see box links under www.vagrantbox.es, however the CentOS might give you only 512M unless you change it in the Vagrantfile after your 'vagrant init'; try 2048 if you have the RAM)

Standards Report

Mark Radford reports the latest from the C++14 Standards effort.

Hello and welcome to my latest standards report. Recently it occurred to me that I haven't (at least for quite a while) given any coverage to how people can involve themselves in the standards process. Therefore, before I get into what's been happening recently, I'll say a little about this. First I'll summarise the standards committee structure and then I'll explain how to get involved.

Regular readers of my reports (and I know for a fact there are at least two of them) will have noticed there are two entities I mention quite a lot: these are the BSI C++ Panel and the ISO C++ Standards Committee. The ISO C++ Standards Committee is the international C++ standards committee, often referred to simply as the 'standards committee' for short. It is comprised of representations from many countries around the world. Each representation comes from the standards body of a member country. The recognised standards body for the UK is BSI, and the BSI C++ Panel represents the UK to the (ISO) standards committee. When the standards committee holds a meeting, the UK delegation is made up of members of the BSI C++ Panel.

If you want to get involved in the standards process, and you are a UK resident, the first step is to join the BSI Panel. Residents of other countries will need to join the equivalent panel/committee of their national standards body (e.g. ANSI in the US). The BSI Panel holds several meetings per year, mostly scheduled around the ISO meetings. The remaining meetings this year are on the following dates: 17 March, 9 June, 21 July, 27 October and 15 December (all Mondays). They will be held at the BSI building, Chiswick High Road, London, starting at 10am and finishing at 5pm (with discussions almost always continuing afterwards in a nearby pub). If you're interested in participating (or think you might be), then please get in touch. The email address is: joining@cxxpanel.org.uk. By the way, you might have heard roles such as the Principle UK Expert (PUKE, for short) mentioned, along with the names of various committees (e.g. IST5), and other such jargon. Don't let that put you off. If you want to get involved you can simply come along to a BSI Panel meeting and participate in the technical discussion without concerning yourself with BSI/ISO committee structure and surrounding jargon. Please be aware that if you want to come along to a Panel meeting you need to get in touch a couple of weeks in advance. This is to make sure you receive a copy of the Calling Notice for the meeting, which (as part of the BSI security procedure) you will need to produce at the BSI reception desk.

It's very much back to C++ coverage this time, as the ISO C++ standards committee is meeting 10th – 15th February (Issaquah, WA, USA). I won't be at the meeting, however I will be keeping up with the updates coming back from that meeting. Unfortunately I will not be able to give the meeting the coverage I would like to, as I am writing this report during the week of the meeting, with the CVu production deadline looming. Steve Love (CVu editor) is very accommodating when it comes to me submitting my reports right up to the production deadline, however this time the deadline is slightly earlier than usual so, sadly, I will only be able to cover the first half of the meeting at the most. Note in passing that the January (pre-Issaquah) mailing is available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/#mailing2014-01.

Moving on to the news I have so far from Issaquah...

First some good news: it was reported after the Monday session, that the C++14 standard is still on track to be shipped this year. Second, some bad news: SG11 (Databases) has disbanded owing to the chair stepping down. So far no one has volunteered to chair this SG, but I'm assuming it could/would reform if anyone did. It's work has not come to an end as it will be continued by the Library Extensions Working Group (LEWG). However, an 'out of the box' database connectivity API is fundamental in modern programming, and is available in popular languages such as Java and C#. Therefore I can only see the demise of the SG dedicated to this area of development as a setback for the development of C++.

The Library Working Group (LWG) spent some time on Monday successfully prioritising all of their 176 active issues. On Tuesday morning the LWG was in joint session with SG3 (see below), and in the afternoon they had a joint session with the Library Extensions Working Group (LEWG). This saw the meeting split into two groups to review the papers that form the inputs to the Library Fundamentals TS. This work continued on Wednesday morning. Meanwhile, the Core Working Group (CWG) have been working their way through their issues list. Later in the week they plan to review the Concepts Lite proposal with the plan of creating a working paper for the planned TS.

Those who follow the activities of SG1 (Concurrency and Parallelism) may be aware of the `shared_mutex` type that was introduced following the Bristol meeting of spring 2013. On Monday SG1 spent some time discussing paper N3891, a last minute proposal (submitted by Gor Nishanov and Herb Sutter) that `shared_mutex` should be renamed `shared_timed_mutex`. Readers interested in the full details can read N3891 for themselves. Briefly though, the rational is that the working paper (N3797) currently specifies three timed mutex types: `timed_mutex`, `recursive_timed_mutex` and `shared_mutex`. N3891 argues for the renaming to avoid a glaring inconsistency. This generated some discussion within SG1 because, although C++14 hasn't yet shipped, some were uneasy about making the change so very late in the day. Another option put forward (during the meeting) was to remove `shared_mutex` from C++14. At the end of the discussion, two straw polls were held to obtain the level of consensus for (1) the renaming option, and (2) the removal option. The results of these polls were very much in favour of renaming and very much against removal. Note that the result of the straw poll does not mean the renaming option will be adopted. That will be decided at the end of the week in a full committee vote (i.e. one vote per national body).

On Tuesday morning there was a meeting of SG3 (File System) in joint session with the LWG. This was to discuss the open issues relating to the File System TS. The bulk of these were comments from the national bodies. As far as the UK is concerned the most substantive issue is the lack of a `relative()` operation to return the relative path. However, another topic of concern was `unique_path()` and an associated potential security vulnerability.

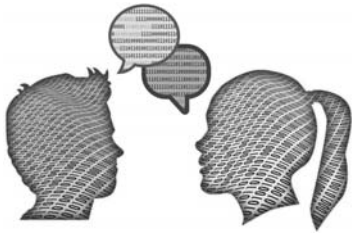
There is evidence that there is a general demand for `relative()`, because it is the most commonly requested extension to the Boost File System library. Note that the TS currently defines what the term 'relative path' means (section 4.18 relative path [fs.def.relative-path]). Two polls were taken: the first was on whether or not `relative()` should be in the File System TS, and the second on whether it should be in the initial version of the TS. Both polls returned a result of very much in favour. When

MARK RADFORD

Mark Radford has been developing software for twenty-five years, and has been a member of the BSI C++ Panel for fourteen of them. His interests are mainly in C++, C# and Python. He can be contacted at mark@twonine.co.uk

Code Critique Competition 86

Set and collated by Roger Orr. A book prize is awarded for the best entry.



Please note that participation in this competition is open to all members, whether novice or expert. 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 trying to write a function to read an integer from a string but I always seem to get zero. Can you advise me?

The code is in Listing 1.

Listing 1

```
#include <algorithm>
#include <iostream>
#include <string>
// Parse integer with optional commas
int readInt(std::string s, int v = 0)
{
  if (s.empty()) return v;
  if (s[0] == ',')
    std::remove(s.begin(), s.end(), ',');
  int digit = s[0] - '\0';
  if (digit < 0 || digit > 9) return v;
  return readInt(s.substr(1), v * 10 + digit);
}
int main(int argc, char **argv)
{
  for ( int i = 1; i != argc; ++i)
  {
    int const v = readInt(argv[i]);
    std::cout << argv[i] << ':' << v << '\n';
  }
}
```

Critiques

Juan Antonio Zaratiegui Vallecillo a.k.a. Zara <zaravalle@gmail.com>

1. The problem lies on the conversion from ASCII digit to integer.

   ```
   int digit = s[0] - '\0';
   ```

   This will give us `digit = ord(s[0])`, e.g. 53 for digit '5'. With digit=48...57 for digits '0'..'9', next line

   ```
   if (digit < 0 || digit > 9) return v;
   ```

   will always return `v`. As the recursive evaluation function is initially seeded with `v=0`, `readInt` will return `0`.

   The simple solution is substituting the first mentioned line with

   ```
   int digit = s[0] - '0';
   ```

   which correctly gives digit=0..9 for s[0]='0'..'9', as required for the simple cases

2. The removal of commas

   ```
   if (s[0] == ',')
     std::remove(s.begin(), s.end(), ',');
   ```

   is deceivingly simple. And wrong.

   For instance, '1,23' is converted into 1233 (at least with my compiler, gcc 4.8. This is UB!). The problem lies on using `std::remove` algorithm, which does not really remove the data, but returns an iterator to the first invalid position.

   The correct line would be:

   ```
   if (s[0] == ',')
     s.erase(std::remove(
       s.begin(), s.end(), ','), s.end());
   ```

   This way, all invalid data is really erased form the string, and '1,23' is converted to 123, as desired.

   But this will only remove all commas, not taking into account any format rule. If we would like the comma to be placed only after thousands and millions... we should probably perform the comma-removal (and validity checking!) prior to entering the recursive function, and removing the comma-removal for inside it.

3. This function takes no caution on possible overflows. Good code design should provide this check.

Hushan Jia <hushan.jia@gmail.com>

Hushan provided a simple correction.

```
#include <algorithm>
#include <iostream>
#include <string>
// Parse integer with optional commas
int readInt(std::string s, int v = 0)
{
  if (s.empty()) return v;
  if (s[0] == ',')
    std::remove(s.begin(), s.end(), ',');
  int digit = s[0] - '0';     // <=====
```


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

Standards (continued)

`unique_path()` was discussed the initial discussion centred around renaming it as the current name does not reflect what it actually does. However the discussion then turned to whether `unique_path()` should be included in the File System TS at all. The result of the poll was very much in favour of removal, so that is what will happen. Note that the File System TS is owned by SG3, therefore SG3 can update this TS to reflect the results of these polls i.e. a full committee vote is not needed.

At this point I have to stop reporting and get this report shipped. Much more will happen this week in Issaquah, but I'm afraid any reporting of it will have to wait until my next report.

```
    // a backslash here mistakenly, which has
    // very different meaning
  if (digit < 0 || digit > 9) return v;
  return readInt(s.substr(1), v * 10 + digit);
}

int main(int argc, char **argv)
{
  for ( int i = 1; i != argc; ++i)
  {
    int const v = readInt(argv[i]);
    std::cout << argv[i] << ':' << v << '\n';
  }
}
```

Paul Floyd <paulf@free.fr>

This is a small example, and there are a couple of easy fixes to get it to work approximately correctly. The first is to change the character literal from `'\0'` to `'0'` in `parseInt`. Secondly, simply `remove()`ing the commas is insufficient. `string::erase()` ought also be called to finish the job.

The next issue that I have with the code is performance. The recursive call to `parseInt` repeatedly calls `remove()` and copies substrings. If the string class has a small string optimization, this might be acceptable. I would recommend:

a) Using erase/remove to remove the all the commas in one go

b) Make the function iterative rather than recursive to avoid making copies. If you really, really want to have recursion, then split the function into two, the first function that strips commas and calls the second, the second being recursive taking either iterators or reference and index to the string rather than copies of substrings.

Unfortunately, this is the tip of the iceberg. Below the water line lurk a host of questions. My main question is 'why remove commas'? Is this to be able to handle either raw numbers and thousands separated numbers? If the latter, then as it stands the code will happily accept ill-formed numbers like "1,,,0". Also hard coding the thousands separator is not a good idea if you want your software to be used outside of the UK and USA. Here in France, the full stop is used as the thousands separator. There are standard functions for converting from strings to integers, such as `std::strtol` and `stoi` (C++11). Unfortunately it doesn't seem too clear to me whether they systematically support thousands separators and locales. Using `stringstream`s might be a better bet, but again there may be issues with locales being well supported. In conclusion, I'd probably try `stringstream`s first, and only then try writing my own function.

Giuseppe Vacanti <giuseppe@vacanti.org>

This code can be made to work by addressing a number of issues:

The call to `std::remove` is not enough to achieve the actual elimination of elements from a container (in this case the character `','`). In this context 'remove' means 'take off from the position occupied' rather than 'get rid of'. Characters matching `','` are moved to the end of the string, and their actual elimination must be achieved through a call to string's `erase`.

After a call to string's `erase` the string could actually be empty, so a new check for the empty string must be carried out.

The character '\0' is not the character corresponding to 0. One should use '0' or the hexadecimal value 0x30.

Something else to remark:

Parsing of the input string stops if a character other than a digit or a coma is found, in particular this means that a negative integer cannot be parsed. I assume this is by design.

The program breaks down if the resulting integer exceeds the maximum value that an `int` can hold. The useful range can be increased by changing to an unsigned integer type (this also makes the inability to parse a negative digit explicit), possibly a longer one (`uint64_t` in c++11, for instance).

I sprinkled a few `const` around, where appropriate, and I stored the value to be returned in a separate variable, as I find that makes checking things in a debugger easier.

With these changes the program (omitted to save space) does what was intended, for instance:

```
cc85>./solution-1 ,0,1,2,3,4,5,6,7,8,9,0,aaa,2
,0,1,2,3,4,5,6,7,8,9,0,aaa,2:1234567890
```

A better solution is presented in the following code where I use the new regular expressions facilities (here via Boost.Regex because my version of gcc does not yet support them, but the syntax is the same), and `_caboost::lexicalst` to catch any overflow.

```
#include <boost/regex.hpp>
#include <boost/lexical_cast.hpp>
#include <iostream>
#include <algorithm>
#include <string>
#include <cstdint>
typedef uint64_t my_int_type;
const boost::regex e("^(\\d+).*");
// Parse integer with optional commas

my_int_type readInt(std::string s) {
  s.erase(std::remove(
    s.begin(), s.end(), ','), s.end());
  boost::smatch match;
  if(boost::regex_match(s, match , e)){
    if(match.size() == 2){
      const std::string new_s = match[1];
      const my_int_type xx =
      boost::lexical_cast<my_int_type>(new_s);
      return xx;
    }
  } else {
    return 0;
  }
}

int main(int argc, char **argv)
{
  for ( int i = 1; i != argc; ++i)
  {
    my_int_type const v = readInt(argv[i]);
    std::cout << argv[i] << ':' << v << '\n';
  }
}
cc85>./solution-2 ,0,1,2,3,4,5,6,7,8,9,0,aaa,2
,0,1,2,3,4,5,6,7,8,9,0,aaa,2:1234567890
```

Marcel Marré and Jan Ubben <marre@links2u.de>

The code contains two bugs, with the first hiding the second initially.

The initial problem is that '\0' in the line

```
int digit = s[0] - '\0';
```

is not the character '0', but the numerical value 0, which means that digit will not be in the correct range for decimal characters. Hence, the function quickly aborts and always returns 0. The corrected line is thus

```
int digit = s[0] - '0';
```

The second problem is undefined behaviour due to the line

```
std::remove(s.begin(), s.end(), ',');
```

Because `std::remove` cannot change the size of a container, it moves later entries further up in the container and any trailing entries are 'valid, but undefined'. In particular, trailing digits could (and do, in gcc-4.7.3) remain and lead to too large a number being returned. For example, calling `std::remove` as above on the string "23,456" could yield the string "234566". The usual idiom is the 'erase-remove' one, where the line becomes

```
s.erase(std::remove(s.begin(), s.end(), ','));
```

With these changes the function will work correctly, but is still far from elegant. Unnecessary recursion with substring copies for each step are both slow and memory inefficient.

A C++ 11-compliant proposal looks as follows:

```
int readInt(std::string const & s)
{
  int v = 0;
  for(auto const c : s)
  {
    if (c == ',') continue;
    int const digit{c - '0'};
    if (digit < 0 || digit > 9) break;
    v = v * 10 + digit;
  }
  return v;
}
```

This has a couple of advantages. It avoids the need for substring copies, and requires no changes to the string, so we can take a const reference as parameter.

Herman Pilj <herman.pijl@telenet.be>

I sometimes wonder whether Roger is the ghost writer behind the code snippets of 'the longest continuously advertised software tool'. [Ed: no, but I do like them!]

I found the `int digit = s[0] - '\0';` quite eye-catching, but there were other problems lurking around the corner.

Each time I encounter a recursive function, the name Fibonacci pops up in my mind. Recursive functions usually look elegant and deceivingly simple. Because the recursion has to stop somewhere, there must be a base or fundamental case where the result is straightforward.

It often leads to compact code, but one of the major drawbacks of recursive code is its execution (in)efficiency.

Let's continue though to look at the code as it is and criticize it (that is what Code Critique is about, isn't it).

First I would like to start to question the way the arguments are passed. I prefer to pass an argument by const reference instead of by value (I strongly dislike the expression 'const reference', it really should be called 'reference to const').

```
int readInt(std::string const & s, int v = 0);
```

As `readInt` is a recursive function the end criterion of the recursion or the fundamental case has to be identified. At first sight, an empty string looks like a good candidate for the end criterion.

Then the code attempts to remove the leading commas. The code to skip the leading commas is the school example of the standard library's `remove` booby trap. How many programmers already have been fooled by the treacherous name of the algorithm `remove` that eventually doesn't seem to remove at all anything from the container. In order to effectively remove the commas, the string needs to be resized.

```
{
  std::string::iterator end =
     std::remove(s.begin(), s.end(), ',');
  s.resize(end - s.begin());
}
```

The code that follows then tries to find out whether the next character is a digit.

The first problem with this code, it that it assumes that the remaining string is non-empty because it tries to dereference the first character of the string.

The second problem is that the code assumes that the digits have consecutive character codes.

This is for sure the case in ASCII and apparently also in EBCDIC.

[I don't know whether Eb is an existing first name but trying to pronounce IBM's character encoding acronym EBCDIC as a word could sound offending.]

The character '0' is not the same as '\0', so that typo ensures that the parsing ends here when you pass real numbers. I wrote a function `runTests` with asserts like

```
assert(readInt("5"), 5);
```

And it coredumps immediately because `readInt` always returns zero because the value of digit is always out of range because of the typo.

In order to find out whether the next character is a digit, you should use the `isdigit` function from `<cctype>` excluding the empty string case. It uses the default locale to find out with the character type facet that the character is a digit.

```
if (s.empty() or not isdigit(s[0])) return v;
```

And then ...

The function then tries to recursively call itself.

I consider that as reasonably dangerous. The recursive function is already too complex. I would rather opt for a recursive function `readDigit`.

```
int readDigit(std::string const &s, int v = 0)
{
  if (s.empty() or not isdigit(s[0])) {
    return v;
  }
  const int digit = s[0] - '0';
  return readDigit(s.substr(1),
    v * 10 + digit);
}
```

I would write the `readInt` as

- `skipWhiteSpaceAndCommas`
- recursively call `readDigit`

Because I wanted to write some C++11 code, I translated skipWhiteSpaceAndCommas into

```
std::string::const_iterator skipIt =
  std::find_if_not(s.begin(), s.end(),
    [](char c)->bool
      {return isspace(c) or c == ',';});
const std::string s1(skipIt, s.end());
```

Here again the character type facet is used to decide which characters are to be considered as space (typically characters like SPC, TAB). The code so far is not able to parse the sign, something like +12 or -54 should be allowed. So I would insert a function `readSign` between `skipWhiteSpaceAndCommas` and `readDigit(s)`.

What about overflows like 12345678901231456?

What about leading zeros?

What about error/exception handling?

Why not disallowing commas, initialize an `istringstream` and just use the extraction operator

```
std::istream& operator>>(int &)
```

And if you then want to use a custom locale, you can simply `imbue` it on the stream.

Commentary

This critique is an example of a common problem – someone trying to implement for themselves a relatively standard function ('read an integer from a string'), but doing it without the breadth of experience needed to do it well.

I think the best answer for the question 'Can you advise me?' is 'Yes – look for an existing solution.'

As two of the critiques mentioned, you can simply use the standard `stringstream` class and, if you want to skip comma delimiters, imbue an appropriate locale. Sadly the current state of support for different locales in standard C++ is less usable / consistent than we might want, but boost offers `Boost.Locale` which does seem to fill in some of the gaps.

Sticking with just standard C++ `readInt` could be re-written as

```
int readInt(char const *locale,
  std::string const &s)
{
  std::istringstream iss(s);
  iss.imbue(std::locale(locale));
  int ival(0);
  iss >> ival;
  return ival;
}
```

and now, depending on locale support in your chosen environment you can choose to handle different conventions for the thousands separators. With Microsoft C++, for instance, this call:

> `readInt("en-GB", "1,234")`
> returns 1234 and
>
> `readInt("es-ES", "1.234")`
> also returns 1234, but – sorry Paul –
>
> `readInt("fr-FR", "1.234")`
> doesn't return 1234 – in this implementation the locale is expecting a 'non-breaking space' character ('\u00a0') as the thousands delimiter.

Using a standard solution means it already deals with issues some people touched on, such as leading - (or +) signs and detecting mis-placed commas.

In my experience the best response to a quite a number of queries is to 'un-ask the question': rather than trying to fix the code the person is trying to write it may be more helpful better to find an alternative (and possibly even a standard) solution.

No-one really tackled the question of error handling – to my mind the contract of this function is inadequate as it simply stops at the first incorrect character and returns what it has so far parsed; with no indication that there were any unparsed characters left.

The Winner of CC 85

This critique attracted several entrants, all of whom correctly identified the 'obvious' fault of using '\0' (although only Marcel and Jan stated what the value of this is – and no-one actually pointed out why this has the numerical value of 0 – which is because it is an octal (!) escape sequence). Many also noticed the incorrect use of **`std::remove`** – an easy mistake to make (and perhaps a good example of a poor name choice.) Fixing this then surfaces the possibility of the string now being empty, which also needs a change to be made.

Giuseppe makes good use of regular expressions in his second solution, which are a powerful technique for string manipulation in C++11, although in this particular case it may be slightly over complicated.

Both Paul and Herman (possibly because of location) both mentioned locales which, while not mentioned by the original problem posed, are likely to be a general thing to bear in mind in today's international world where the users of your programs may well be using a different language and conventions from your own. They also both mentioned the standard facilities provided by **`stringstream`**s and locales for handling the job of converting a string into an integer.

In my opinion Herman provides a slightly fuller critique of the original problem and a good explanation of things to think about in general with recursive algorithms so I have awarded him the prize for this issue's contest.

Code Critique 86

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

> This code works for me but not for my co-worker. I get
>
> `31-Mar-2013 01:30:00`
>
> and they get:
>
> `java.text.ParseException: Unparseable date: "2013-03-31T01:30:00-04:00"`
>
> What's wrong with their machine? I can't easily check as they're in a different country!"

Can you solve the mystery and identify any other reasons why the code might be problematic?

The code is in Listing 2.

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.

Listing 2

```
import java.text.*;
import java.util.*;
public class DateTimeTest
{
  static String format =
    "yyyy-MM-dd'T'HH:mm:ss-SSS";
  public static void main(String[] args)
  {
    SimpleDateFormat sdf =
      new SimpleDateFormat();
    sdf.applyPattern(format);
    sdf.setLenient(false); // Require 2-digits
    testParse(sdf);
  }
  static void testParse(SimpleDateFormat sdf)
  {
    Date then = new Date();
    try
    {
      then = sdf.parse(
        "2013-03-31T01:30:00-04:00");
    }
    catch (Exception ex)
    {
      System.out.println(ex);
      System.exit(1);
    }
    DateFormat df =
      DateFormat.getDateTimeInstance();
    System.out.println(df.format(then));
  }
}
```

View from the Chair

Alan Griffiths
chair@accu.org

I need to bring two issues to your attention. They seriously affect the running of the organisation and need you to act.

The first issue is the make-up of the committee. There are only seven posts for which candidates are standing at the election. (You can see details on the members area of the website.) None of the posts is being contested so, barring the unforeseeable, we can predict next year's committee will be the candidates that have stood.

Seven committee members is a worryingly small number (there are fifteen currently listed). A committee of seven means that three members including an officer is a quorum. In addition, two of our 'officer' posts will be vacant. These are the Chair and Secretary.

The constitution requires us to have four officers and something must be done.

The first thing that can be done about this is to invoke clause 5.3.4 of the constitution:

> If no candidate for an officer position is nominated according to the procedure in 5.3.3, nominations for a caretaker to fill the vacancy can be taken from the floor. The duty of a committee with at least one caretaker officer will be to organise a new election for that role. In the meantime, that committee will be limited to ordinary administration of the organisation.

During the AGM someone can be appointed as a 'caretaker' Chair and/or Secretary. They don't have to be present but as this is one of the few cases where things can happen at the AGM without prior notice to the membership it would be helpful if the presiding member can validate their willingness to stand at (or prior to) the AGM. Note that such an appointment would only be until the committee can organise an election.

The elected committee can also co-opt additional members to the committee for vacant roles. This comes under two parts of the constitution:

> 5.4.1 Should any member of the Committee resign or cease to act during the life of the Committee, or a vacancy otherwise arise, the Committee shall have the power to co-opt a member of the Association to fill the vacancy.
>
> 5.4.2 The Committee shall have the power to co-opt any member of the Association for a particular service. Co-optees shall have voting rights on matters pertaining to the service for which they were co-opted.

This power is normally used to co-opt people to roles with special requirements (such as the Conference Chair). It could, in theory, be used to fill posts left vacant at the AGM but doing so would leave the membership with no say in an important decision.

In view of the above, the incoming committee will be meeting after the AGM in order to deal with any co-options. The details are not confirmed at the time of writing, but they will likely be in one of the smaller conference rooms during the afternoon break. If you are willing to help out in some way, please attend (or, if you can't attend, let them know).

The second matter is the 'hardship fund' the ACCU has been maintaining to support the memberships of individuals who could not finance themselves. However, there have been decreasing calls on this fund over time and nothing has been paid out for some years. (Many of the original reasons for 'hardship' – such as difficulty converting currency – are less frequent than they were.)

The approximate balance of this hardship fund is £2,000.

The view of the committee is that this fund no longer serves its original purpose. We've asked for suggestions but none have been forthcoming.

I'm proposing a motion at the AGM to discontinue the 'Hardship fund' and roll this money into the general finances. If this is accepted committee can still allocate funds to hardship cases if and when they occur.

Bookcase

The latest roundup of book reviews.

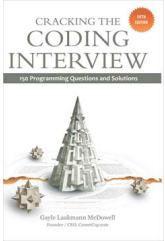
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

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-glamorous 'not recommended' rating, you are entitled to another book completely free.

Thanks to Pearson and Computer Bookshop for their continued support in providing us with books.
Jez Higgins (jez@jezuk.co.uk)

Cracking the Coding Interview

By Gayle Laakmann McDowell, ISBN: 978-0-9847828-0-2, published 2012, 498 pages (of which >300 are answers to questions)



I bought this book the last time that I was looking to change jobs. In the event, I changed jobs before I had time to read it. Now I'm on the other side of the interviewing table and I thought that it might be a source of ideas.

Overall, I found the general advice to be sound. Prepare yourself, know your target company and some background on large US-based multinationals like Google, Apple, Microsoft and Yahoo.

The technical questions and answers were, I felt, rather unexceptional. Mostly the code is Java, with the aim of being easy to understand. In particular in the Bit Manipulation chapter I kept on saying to myself "that won't work reliably in C or C++". In the chapter dedicated to C and C++ (which is almost entirely C++), several things shocked me

1. use of `#define` for constants
2. use of raw pointers and even `malloc`/`free` (though somewhat predictably, the pointers segue into virtual functions); I would expect candidates to show a better understanding of the use of the heap and the stack and for me systematic overuse of `new` is java++ – C++ written in a Java dialect rather than idiomatic C++.
3. no mention of exceptions
4. no mention of C++11
5. question as to why derived class destructors need to be virtual.

One last sin, it's all a bit computer science-y (as Press *et al*. might say). There's a lot on algorithms, recursion and even databases, but when it comes to a tiny bit of numerical analysis, I have the impression that the author is lost at sea. The example in question is oriented towards data structures used to evaluate a "mathematical expression ...Ax^a + Bx^b ...". I suspect that a polynomial is intended, but the text does state that the exponents are of type double. No mention of the requirements for rapid and accurate evaluation of the expression. Who cares if the answer is NaN when the inputs are wrapped in a nice class!