overload 103

JUNE 2011 £3

Some Objects Are More Equal Than Others

We see how languages can treat object equality differently, with many subtle pitfalls

Systems Thinking Software Development

Applying a Systems Thinking approach can improve your software process

The Guy We're Working For

Developers and their users *can* co-exist happily! Here's how...

Interval Arithmetic Won't Cure Your Blues

Our last forray into the perils and pitfalls of floating point calculation

Exception Specifications in C++ 2011

We present an in-depth look at the new C++ exception specification mechanism

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 103

June 2011

ISSN 1354-3172

Editor

Ric Parkin
overload@accu.org

Advisors

Richard Blundell
richard.blundell@gmail.com

Matthew Jones
m@badcrumble.net

Alistair McDonald
alistair@inrevo.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simon.sebright@ubs.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 104 should be submitted by 1st July 2011 and for Overload 105 by 1st September 2011.

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

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.

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

Can you keep a secret?

Privacy and security have been in the news a lot recently. Ric Parkin looks behind the curtain.

It's not been a good few months for Sony.

First of all it was one of many companies whose manufacturing plans were thrown into turmoil due to a major earthquake and tsunami. It wasn't so much the direct damage, but disruption to power generation and supply chains has shown how vulnerable Just In Time production methods are to even small delays. [Sony]

The automotive industry was affected even more, as it turned out that a single chip making plant that was destroyed made about 40% of the chips used worldwide in car manufacturing. With deliberately low stocks of parts, car production has been severely disrupted [Renesas]. This did make me wonder what sort of equivalent risks to production applied to software development, given that the Toyota Production System, and other JIT processes are the inspiration behind many Agile development practices. A few spring to mind – a major risk is an unexpected change in production capacity. This will usually be caused by personnel changes, such as illness or leaving the company. Finding a replacement and getting them up to speed is a non-trivial effort, which is why Brooks's Law was noted [Brooks]. Less serious causes can include power cuts, and problems with computers and networks.

As I write this, the PlayStation network has only been partly restored following an intrusion that potentially exposed personal details of millions of users. Unfortunately they intially turned a serious problem into a PR disaster by looking to be slow to admit to the problem, or giving details about what had actually been compromised. Some of this could well have been due to the difficulty of tracing where exactly the intrusion had access to, what had been taken, and how this would affect users. But some things were definitely handled badly, in particular whether passwords had been stored in plain text or not. It turned out they had correctly only stored a hash – a large number or string that was generated from the password and used to confirm you've typed a password in correctly without actually transmitting or storing the password itself [Hash], but it took time to clarify this.

Unfortunately, some identity information was stored such as dates of birth, and it's this that is the main cause of concern as it can be used as the basis of identify theft. It has severely dented their reputation. It did make me question my own approach to computer and identity security, both as a developer (yes, we only store hashes!) and as a user. I also recently updated our password dictionary for cracklib [Sourceforge], which sees how secure a new password is. The new dictionary is massively larger than our previous version, and we're finding that it includes many passwords that used to be thought of as strong, but now appear in dictionaries that are used by brute-forcing algorithms. I'm seriously reviewing my password policy to make them harder to guess, and will avoid supplying unnecessary personal details (I was always reluctant anyway).

Other sources of leaked information have been making the news recently. One high profile one was finding out that iPhones stored a list of locations where you'd been [Jones]. While this was only used internally for some performance improvements, again it worried a lot of people as it effectively gave anyone with access to your phone (or the iTunes backup on your computer) a log of your movements. Promptly a fix has been issued to delete the data when no longer needed. Of course, the authorities have other ways of tracking your phone, even if they're too much effort to deploy except in serious cases. The obvious one is that phone companies have logs of which mobile stations you can connect to, which is enough to track you fairly accurately via some simple triangulation. Even before that, getting access to telephone logs and doing some fairly simple traffic analysis could be used to pick up patterns, and reveal the structure of an organisation. I was reminded of this recently after seeing the latest incarnation of some of this analysis software [i2]. Most scary of all is that I was heavily involved with a major rewrite of this software back in the mid to late 90s, and it was interesting to see that despite over a decade of improvements, there were still signs obvious to me that the core of my code is still there. A warning that code can last for longer than you might think!

Of course, if an identity thief wanted a your details, or someone wanted to track you, it's probably easier to just keep an eye on people's Facebook updates and pictures in the Cloud. It's troubling just how much personal information can be gleaned by even a cursory glance and some simple searches, and when coupled with position updates via tools such as Foursquare, it's pretty easy to see where someone is, what they are doing, and who else was there. If that's in real time and the person's address is known, and a break-in would be trivial.

The other interesting technology news thats been around recently is the way that Twitter is being used to get around so called 'Super-injunctions' and reveal secrets that people had been trying to keep under wraps. By rapidly retweeting, a story be spread extremely quickly, and the 'Spartacus effect' of thousands of people doing it makes them think that they are immune from prosecution. Time will tell whether that will remain true, as there's already talk of disclosing details to the police of people who have helped. Dispiritingly, most of the cases seemed to be celebrities trying to conceal affairs, which is a sad reflection on certain sections of the press. Personally I'm not interested in that at all – they have as much right to a private life to muck up as I do. But there are issues of privacy, freedom of speech, and a society fast changing how it communicates.

One thing that was worrying though – at one time someone posted a list of supposed injunctions which turned out to be wildly innacurate (some so bizarre you just knew they were a joke), causing some swift rebuttals and embarrassment. Is this a taste of things to come, where fast

Ric Parkin has been programming professionally for around 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he's left a trail of new members behind him. He can be contacted at ric.parkin@gmail.com.

communications and ‘Chinese Whispers’ cause all manner of wild stories and accusations to be propagated? As the saying goes, a lie is halfway around the world before the truth has got its boots on. In this vein, there was an interesting experiment performed accidentally by Graham Linehan who writes the sit-com *The IT Crowd* [Linehan]. After tweeting an amusing lie – that Bin Laden was watching the show on the captured videos – he was suprised just how fast it spread and mutated incorporating completely random stuff, before he finally exposed it.

And sometimes people just won’t talk about it when you want them to – I noticed a couple of comments recently from ACCU developers who’d written their own iPhone games about how much effort it was to try and generate some interest. With so many apps to choose from it’s now an uphill struggle to get any attention.

Bubble 2.0?

We seem to be in a technology stock bubble again. Things that make me feel this way include the recent purchace of Skype by Microsoft for a massive $8.5bn, and the imminent floatation of LinkedIn at a large valuation, and rumours about FaceBook or Twitter being floated soon. It all feels very reminicent of 2001, although this time it’s social networking driving interest instead of early internet companies and biotech. But yet again to pick the real winners without over paying for them will be hard, especially when what seems to be the next big thing suddenly goes out of fashion, or more likely, becomes so widespread it’s no longer what makes a company unique and hence valuable. Buyer beware.

References

[Brooks] http://en.wikipedia.org/wiki/Brooks's_law

[Hash] http://phpsec.org/articles/2005/password-hashing.html

[i2] http://www.bbc.co.uk/news/uk-13366706

[Jones] http://www.bbc.co.uk/blogs/thereporters/rorycellanjones/2011/04/iphone_tracking_creepy_cool.html

[Linehan] http://www.bbc.co.uk/news/magazine-13467407

[Renesas] http://www.bbc.co.uk/news/business-13421065

[Sony] http://www.bbc.co.uk/news/business-13557431

[Sourceforge] http://sourceforge.net/projects/cracklib/

Bletchley Park fundraising effort

The past two Novembers have seen the enjoyable ACCU Security conferences, held at Bletchley Park to raise money for their activities. Well, Astrid Byro has decided to go that extra mile this year to raise even more. About three and a half miles to be more accurate – upwards. On 16th August she’s going on an 8-day trek to the Everest Base Camp, which is 5,545 metres above sea level. ‘You must understand the context of this endeavour.’ she says. ‘ I’m afraid of heights and this will challenge my fears on a daily basis with multiple crossings of rickety bridges across torrential gorges. In addition, I will be doing this at the end of monsoon season so there is the ever-present danger of flash floods as well as the menace of leeches. I hate leeches.’

She’s set a fundraising target of £50,000, so would be a great help to Bletchley. She is hoping to achieve this target by donations as well as corporate sponsorship so if you would like a photo of your corporate logo flag flying at Base Camp, want her to wear sponsored logo clothing, or you have a stunt in mind, she’s open to negotiation.

You can follow Astrid’s progress on her blog as she pursues her training programme, at www.abc-ebc.blogspot.com and you can support her by making a donation at www.justgiving.com/Astrid-Byro . Good luck!

[Photograph published under Creative Commons Licence 3.0 – original can be found at http://www.happytellus.com/gallery.php?img_id=5143]

Some Objects Are More Equal Than Others

Comparing objects is a fundamental operation. Steve Love and Roger Orr consider different language approaches.

Testing for equality is an important concern in a lot of programming tasks and is often used for control flow: equality is one of the commonest expressions used in **if**, **for** and **while** statements. However despite being something that is covered in almost any introduction to a programming language the concept and implementation of equality can be quite complicated.

Possible meanings of 'equality'

There are a wide variety of meanings to the use of 'equality' in a programming language. The list of possible meanings includes:

1. Refer to the same memory location
2. Have the same value
3. Behave the same way

This article explores some of the details and pitfalls with equality in terms of just the first two items on this list. We found it was a harder task than it appears at first glance to get it right (for some definition of right), even ignoring the third item on our list or looking further afield for other meanings.

The first item in the list is often described as 'identity comparison' and the second one as 'value comparison', and we make use of these terms below. Note that value comparison usually refers to the *perceived* value for users of the object and fields that don't affect this (for example internally cached values) are usually not included in the comparison code.

We are further restricting the subject to focus primarily on only three languages: C++, C# and Java. Despite their common heritage and obvious similarities there are many differences in the sort of problems equality raises in each language: even at the basic level of language syntax we see:

- In Java: **a == b** always does something for all variables **a** and **b** of the same type (and compiles in some cases when they are of different types) and you cannot change what it does.
- In Java & C#: **anobject.[eE]quals(another)** always does something (we write **[eE]quals** because the method is spelled **equals** in Java but **Equals** in C#)
- In C++ & C# you can overload the meaning of **==** and in C# & Java you can override **[eE]quals** to customize behaviour.

Let's start with the language construct form of equality '**==**' on the grounds that this must be a pretty fundamental definition to have been enshrined in the syntax of the programming language, What does each language provide for this operator 'out of the box'?

In C++ '**==**' is predefined (as a value comparison) for all built-ins and the subset of the library types for which equality makes sense (e.g. **std::string**), but is not automatically provided for custom types defined in a program. However you can provide your own definitions of **operator==** as long as at least one argument is a custom type: and you can also specify your own return type for the operator (although returning anything but **bool** is usually a bad decision.)

In Java '**==**' is predefined for primitive built-ins and does a straightforward value comparison. For object types '**==**' performs identity comparison between the two objects supplied. You cannot change this behaviour.

In C# '**==**' is predefined, or overridden, for all built-ins and library types (whether these types are *reference* [**class**] or *value* [**struct**] types). It is not automatically provided for custom value types and performs identity comparison for custom reference types. C# lets you define '**==**' for any custom type, but you must additionally provide an implementation of '**!=**'.

Object comparisons

For Java and C# the presence of a single root class for all object types allows for a sensible definition of an equality method in this base class which takes an argument of the base class. In both languages the default implementation of this method, on custom types, performs identity comparison.

Java overrides **equals()** for some of the predefined types, such as **Integer**. However there is some confusing behaviour as Listing 1 demonstrates.

If you compile and run this simple program you might be surprised:

```
Testing 10
Equals
==
Testing 1000
Equals
```

```
public class IntegerEquals
{
  public static void main(String[] args)
  {
    test(10);
    test(1000);
  }
  public static void test(int value)
  {
    System.out.println("Testing " + value);
    Object obj = value;
    Object obj2 = value;
    if (obj.equals(obj2))
       System.out.println("Equals");
    if (obj == obj2)
      System.out.println ("==");
  }
}
```

Listing 1

Steve Love is a programmer who gets frustrated at having to do things twice. He can be contacted at steve@arventech.com

Roger Orr has been programming for far too long but still enjoys it far too much. Some of it is paid and some of it isn't. He can be contacted at rogero@howzatt.demon.co.uk

unit tests, which typically use compile time strings, will pass most tests successfully

```
public class Intern
{
  private static final String s1 = "Something";
  private static final String s2 = "Some";
  private static final String s3 = "thing";
  public static void main( String [ ] args )
  {
    if (s1 == s2 + s3)
       System.out.println("match!");
  }
}
```

Listing 2

The two objects `obj` and `obj2` compare the same using the `equals` method (as the overridden method in Integer compares values not identity) when executed with `value` set to 10 or 1000 as expected. However, on most implementations of Java, `obj` compares the ***same*** as `obj2` using `==` when value is set to 10 but they compare ***different*** when the value is set to 1000. What is happening here?

This is a consequence of an optimisation in the Java code that boxes primitive data into Integer objects. The compiler implements `obj == value` by calling `Integer.valueOf(value)` and this method caches 'commonly used values' such as 10[1]. Hence in the first case the compiler is performing identity comparison on two references to the same, cached, `Integer` with value 10 and in the second case the compiler is performing identity comparison on references to two *separate* temporary `Integer` objects with value 1000.

There is a similar problem with intern'ed strings (strings held in a shared pool of unique strings normally accessed using the `String.intern()` method) as demonstrated in Listing 2.

This program prints `match!` when executed as the compiler ensures that strings with the same compile time value generate references to a single object. This is perfectly safe since strings in Java are immutable, but can cause some confusion. In general checking strings for equality in Java with `==` is unsafe and some tools provide a warning for attempts to do so. The danger is that compile time strings, which are interned, are treated differently from any runtime strings (which typically aren't).

The classic case where this causes problems is that unit tests, which typically use compile time strings, will pass most tests successfully whether you use the `equals` method or the `==` operator; but in actual use with runtime generated strings (such as those read from a file) the behaviour is different.

C# implicitly provides some implementation assistance with the `Equals` method for value types, but it's more complicated than it might appear at first sight. (Listing 3)

```
struct Easy
{
  int X;
  int Y[ 100 ] ;
}

struct Hard
{
  int X;
  MyType Y;
}
```

Listing 3

Both these structures will have an `Equals` method synthesised by the compiler. The first class (`Easy`) only contains basic scalar members and the `Equals` method will perform a bit-wise check on the two values (using the total size of the object), which is often exactly the desired behaviour (and is fast). In the case of the second class (`Hard`) the presence of the custom type `MyType` means that the synthesised `Equals` method performs reflection on the class at run time to identify the fields and then does a member-wise comparison of all the members (including the basic scalar `int` member `X`). While this produces the correct answer the performance is likely to be significantly worse than an explicit implementation of equality.

Finally in both C# and Java thought needs to be given to ensure the primary object reference is non-null. The simple example in Listing 4 demonstrates the problem and also a way (in C# only) to avoid it.

This program fails with a `NullReferenceException` as `a` is null in the first call to `Equals` and you cannot call a method on a `null` object. The second call, using the static method taking two arguments. does not throw such an exception when supplied with null references (and returns `false` if either `a` or `b` is `null` and `true` if they both are).

C++ does not have a single object root and so it doesn't really make sense to have an equals method, but it ***does*** have templates and to help with programming the STL there is `std::equal_to`, which by default performs `==`. You can specialise it for your own type to pass your own

```
public class NullEquals
{
  public static void Main()
  {
    object a = null;
    object b = new object();
    if (a.Equals( b ))
      Console.WriteLine("Now there's a thing");
    if (object.Equals(a, b))
      Console.WriteLine(
      "This should be safe enough");
  }
}
```

Listing 4

1. See http://download.oracle.com/javase/6/docs/api/java/lang/Integer.html#valueOf%28int%29

The trouble is that the overloaded method is called based on the compile time type of both the primary object and the argument

```
#include <functional>
#include <iostream>
int main( )
{
  std::cout << "std::equal_to<int>()(10,10): "
    << std::equal_to<int>()(10,10) << std::endl;
}
```

Listing 5

types to methods and classes implemented in terms of `equal_to` such as `std::unordered_map` (see Listing 5 for an example).

The full story for C# is even more complex as there is a long list of equality measures, which have been added to as various new versions of the .Net framework have been released. The list includes:

- `object.Equals` (we've already seen both flavours of this one)
- `object.ReferenceEquals`
- `IEquatable<T>`
- `IEqualityComparer`
- `IEqualityComparer<T>`
- `EqualityComparer<T>`
- `IStructuralEquatable`
- `StringComparer`

...and others we've probably missed...

The second element of this list, the `ReferenceEquals` method, is used to perform the identity check: that two references refer to the same object. The method is needed because `==`, which performs this check by default, can be overridden. (Since Java does not allow operator overloading it has no need for such a method.)

However, when used in conjunction with object boxing, `object.ReferenceEquals` has some interesting behaviour (see Listing 6).

This program prints `False` because the two temporary boxed integer objects created to pass into the `ReferenceEquals` method are distinct, and hence different, objects. This is a related problem to the one shown above using the Java `Integer` class.

```
public class RefEqual
{
  public static void Main()
  {
    int ten = 10;
    System.Console.WriteLine(
       object.ReferenceEquals(ten, ten));
  }
}
```

Listing 6

Overloading equality

Both Java and C# allow the programmer the freedom to overload the `[eE]quals` method to take argument of a different type. Listing 7 is an example in Java that shows the problems of a naive implementation.

This program prints:

```
oe1.equals(oe2): true
oe1.equals(obj2): false
obj1.equals(oe2): false
obj1.equals(obj2): false
```

even though the **same** objects are being compared in each case. The trouble is that the overloaded method is called based on the **compile** time type of both the primary object and the argument. What you probably want in this case is logic based on the `runtime` type.

```
public class OverloadingEquals
{
  private int value;

  public OverloadingEquals(int initValue)
  {
    value = initValue;
  }

  public boolean equals(OverloadingEquals oe)
  {
    return oe != null && oe.value == value;
  }

  public static void main(String[] args)
  {
    OverloadingEquals oe1
       = new OverloadingEquals(10);
    OverloadingEquals oe2
       = new OverloadingEquals(10);
    Object obj1 = oe1;
    Object obj2 = oe2;
    System.out.println("oe1.equals(oe2): "
       + oe1.equals(oe2));
    System.out.println("oe1.equals(obj2): "
       + oe1.equals(obj2));
    System.out.println("obj1.equals(oe2): "
       + obj1.equals(oe2));
    System.out.println("obj1.equals(obj2): "
       + obj1.equals(obj2));
  }
}
```

Listing 7

Our problems are mostly caused by attempting to define equality in a class hierarchy

There are some principles from the mathematics of 'equivalence relations' that, if adhered to, result in a consistent use of the concept of equality. They are that equality is...

- **Reflexive**

  a==a is always true
- **Commutative**

  if a==b then b==a
- **Transitive**

  if a==b and b==c then a==c
- **Reliable**

  Never throws. (This means checking for null!)

These rules are listed out in fuller detail in the language references for both C# [C# Equals] and Java [Java equals]. The wording from the C++ standard is short enough to quote in full: '(5.10p4) Each of the operators shall yield true if the specified relationship is true and false if it is false.' There you have it: succinct at any rate!

Now let us try and apply these rules when considering polymorphic equality. Consider a two-dimensional coordinate class in C# (Listing 8).

We might extend this class to support a three-dimensional coordinate system (Listing 9).

How does this polymorphic equality fare when checked against our four relationships for equality?

```
var p1 = new Coordinate { X = 2.3, Y = 5.6 };
var p2 = new Coordinate3d { X = 2.3, Y = 5.6,
                            Z = 10.11 };

p1.Equals( p2 ) is True
p2.Equals( p1 ) is False
```

Oops. The equality relationship fails the ***commutative*** requirement. We can improve our conformance to this requirement in C# by implementing `IEquatable<T>` – which enforces implementation of an override of `Equals` taking `T` – for both classes. This provides the symmetry for `p1` and `p2` but is still not a complete solution to the problem as this code fragment shows:

```
object o1 = p1;
Console.WriteLine(
   "p1.Equals(o2) {0}, o2.Equals(p1) {1}",
   p1.Equals(o2), o2.Equals(p1));
```

However, even if we fix the commutative relation by making our equality test more complex we *still* have a problem. Let's add this variable:

```
var p3 = new Coordinate3d {
   X = 2.3, Y = 5.6, Z = 1.22 };
```

Now `p1` will be equal to `p3` (for the same reason it is equal to `p2`), but `p2` and `p3` will *not* compare equal. We have broken the ***transitivity*** requirement. How can we resolve this? Should we even try?

Let's consider *why* we have the problems we see. Our problems are mostly caused by attempting to define equality in a class hierarchy. What sense is there to try and compare a two-dimensional and three-dimensional object? They are not the same class. The first solution is to change our design so that two and three dimensional classes are not related: we might use composition in preference to inheritance if we do wish to use some of the implementation of `Coordinate2d` in the implementation of `Coordinate3d`.

When inheritance is needed a good solution to the problematic elements of value equality is to allow comparison to succeed only if the actual run-time class types are the same, which can be implemented simply enough in C# by comparing the results of calling `GetType()` on each object.

```
class Coordinate
{
  public double X { get; set; }
  public double Y { get; set; }
  public override int GetHashCode()
  {
    // ...
  }
  public override bool Equals(object other)
  {
    var right = other as Coordinate;
    if (right != null)
      return X == right.X && Y == right.Y;
    return false;
  }
}
```

Listing 8

```
class Coordinate3d : Coordinate
{
  public double Z { get; set; }
  public override int GetHashCode()
  {
    // ...
  }
  public override bool Equals(object other)
  {
    var right = other as Coordinate3d;
    if (right != null)
      return base.Equals( other ) &&
        Z == right.Z;
    return false ;
  }
}
```

Listing 9

it may make the **initial implementation simpler** to define **'just enough' equality** to be able to use the type in this way

Incidental and intentional equality

Avoid defining equality just so it can be used in conjunction with something that requires it, e.g. hashed containers. While it may make the initial implementation simpler to define 'just enough' equality to be able to use the type in this way, such partial implementations of equality have a nasty habit of causing more serious problems later on as the code evolves.

Suppose for example that you have a C# class and wish to create a `HashSet` of objects from this class. It can be tempting to define an `equals()` method on the class that fulfils just the checks necessary for this usage. However the equality used for a comparison in this context might be very different from one used elsewhere: perhaps only certain key fields are relevant. In this case an alternative way of solving the problem exists as the C# `HashSet` can use a pluggable equality comparer (`IEqualityComparer<T>`) instead of using `equals()`. This also provides a clearer way of stating the intent than implementing the equality operator just for using in the hash set. In C++ the `unordered_set` can be given its own equality comparer; however in the standard Java collection classes `HashSet` can only use `object.equals()`, so you're stuck with it.

Within a single application, *both* meanings of equality might be required: for example in an application for playing card games do you need **the** Ace of Spaces or **an** Ace of Spades? In Java and C#, override `[Ee]quals` for a value-check and leave `==` well alone to perform its default action of an identity check. In C++, which allows access to the address of an object, you can explicitly compare addresses (for identity) or contents (for value). Unless of course someone has defined `operator&` for one of the types...

Hashcodes

There is a close relationship between equality and hashing. For example the C# documentation states that 'classes [..] must [...] guarantee that two objects considered equal have the same hash code'. Java imposes a similar rule for `Object.hashCode()`.

The reason is simple: when hashing functions are used with collections of objects the hash code is used first as a coarse filter to partition objects into buckets with the same, or related, hash codes. If you implement a hash code function that means two objects comparing equal have a different hash code then the two objects may end up in different buckets and the code won't ever get to the point of testing for equality.

Hash codes for objects that can mutate are another problem. See Listing 10 for an example.

Consider what happens if `Value` changes after inserting into a hashed container... if the object's hash code changes after being added to a hashed container, subsequent attempts to look for the object in the container will be accessing the wrong bucket.

The default implementation of `GetHashCode()` in C# for a value type is the hash code of the first field – this is rarely the best implementation for most value types. While we were investigating hash code behaviour in C# we found an interesting 'feature' of the Microsoft C# runtime: the hash code for a `boolean` value is constant! The program in Listing 11 demonstrates both these behaviours by printing `True` both times when compiled and run using Microsoft's implementation.

```
public static class Bogus
{
  public String Value;
  @Override public int hashCode()
  {
    return Value.hashCode();
  }
  @Override public boolean equals( Object other )
  {
    return ((Bogus)other).Value.equals( Value );
  }
}
```

Listing 10

Using Visual Studio this program prints:

```
True
True
```

Collections

Another set of issues is raised by considering equality on container types. When are two collections of things equal? Is it enough that the two containers have the same items or do they need to be in the same order? (As a side note, we can add to the C# list of equality checks with `SequenceEqual`, which insists on the same items, in the same order).

```
using System;
static class Program
{
  struct HashTest
  {
    public bool Enabled;
    public string Value;
  }

  public static void Main()
  {
    var h1 = new HashTest{
       Enabled = true, Value = "Great!"};
    var h2 = new HashTest{
       Enabled = false , Value = "Great!"};
    Console.WriteLine(
       h1.GetHashCode() == h2.GetHashCode());
    h1.Value = "Rubbish!";
    Console.WriteLine(
       h1.GetHashCode() == h2.GetHashCode());
  }
}
```

Listing 11

do two containers match if they **contain the identical objects** or if they contain objects with **identical values**?

This is a question that has performance implications too: comparing two sets are equal when permutations are allowed has a higher complexity measure than the case when the ordering must match.

A further question that may need addressing with containers is whether you want a value or reference comparison: do two containers match if they contain the identical objects or if they contain objects with identical values?

Note that this is a case where polymorphic equality makes a lot of sense: two collections are equal when they contain the same objects. You are not usually interested in whether they are from the same class (or even whether the internal states are the same); the important thing for equality is the objects they contain.

Conclusion

Equality is hard to define simply even for a single language. It is easy to implement if you stick to a small set of common sense rules; more complicated implementations are possible but not in general recommended.

One key distinction is between values and references. You should know the difference between (polymorphic) reference types and value types in all languages and avoiding treating the two the same way! Equality for references is a check for identity but equality for value types is a check for equal values of all (significant) fields.

Making use of immutability for value types has many benefits, far beyond equality. In the case of equality though it allows for the possibility of caching of objects and/or values and it also removes the class of problems exemplified by the example of modifying an object while it is held in a collection.

Using value equality in a class hierarchy rarely makes sense and should be avoided. It is often better to avoid inheritance in the sort of cases where equality might make sense and use composition instead. Classes can also be made `final` (or `sealed`) to prevent unwanted inheritance but this can be an annoyance when a user of the class has a valid reason for wanting to extend your class. ■

Further reading

C# in a Nutshell has a deep exploration of equality in C#. For more about equality in Java see http://www.javapractices.com, and follow links through Overriding Object methods to implementing equals.

Angelika Langer and Klaus Kreft wrote a pair of articles on the subject [Langer]. While the target of their article is Java many of the points apply to C# as well.

References

[C# Equals] http://msdn.microsoft.com/en-us/library/bsc2ak47%28v=VS.100%29.aspx

[Java equals] http://download.oracle.com/javase/6/docs/api/java/lang/Object.html#equals%28java.lang.Object%29

[Langer] http://www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html

The Guy We're All Working For

Developers like to think they're in control of their products. Sergey Ignatchenko reminds us who's really in charge.

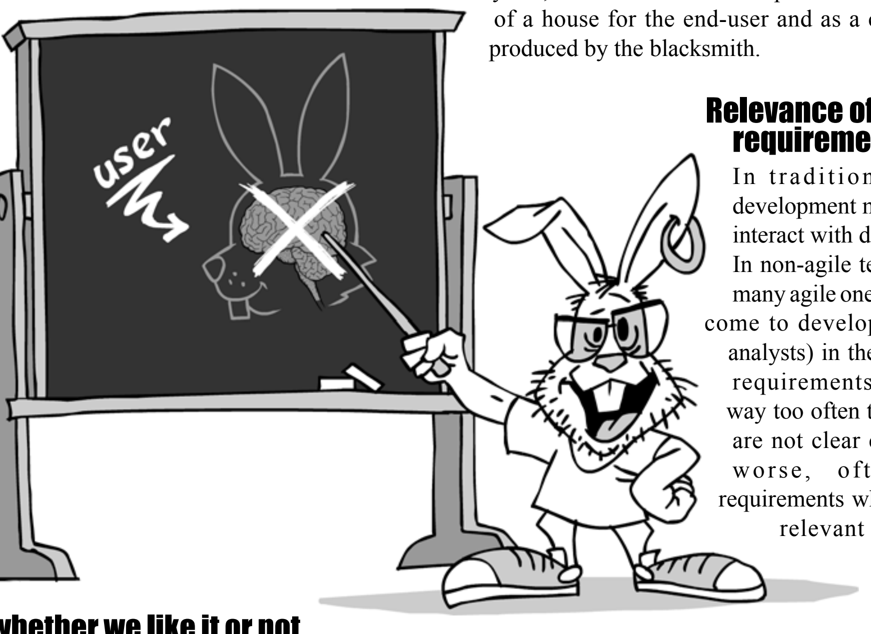
Disclaimer: as usual, opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with opinions of translator and *Overload* editors; please also keep in mind that translation difficulties from Lapine (like those described in [LoganBerry]) might have prevented from providing an exact translation. In addition, both translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

Laynt Preenahlarny naylte vao aisi nao?

Was Laburnum a good or bad rabbit?

Users and developers a.k.a. Elil and Naylte

Program users and program developers are two camps which are traditionally not that fond of each other (to put it mildly). Users tend to think that developers are stupid idiots doing nothing more than intentionally inserting bugs into the programs; advanced users are often even more annoying to developers, arguing that certain features (the ones they want) can be added without any problems in two days (whereas from the developer's perspective it will take two months and will break a dozen other features that millions of other users rely on). Developers, on the other hand, tend to forget about users at all, and if forced to speak on this subject, will rarely characterize users any better than 'mindless creatures without brain or purpose'[1].



The user has the upper hand, whether we like it or not

On the surface, it may seem that this mutual dislike between users and developers is symmetrical in nature, but in fact it is not. If the users don't value the product (in whatever way *they* define value), they won't use it and the whole project will be a failure. And as it is the *user* who eventually decides if the project is successful, the relationship between users and developers is an inherently asymmetrical one, with users having the upper hand. Obviously developers have the option to ignore users, but in a modern economy if suppliers (in our case – developers) don't have a monopoly and ignore the needs of their consumers (in our case – users), the chances of success of the supplier/developer become infinitesimally small. In a market economy suppliers exist for *only* one purpose – to satisfy the needs of their consumers, and if the supplier ignores these needs – it dies, usually sooner rather than later.

Here I need to mention that for the purposes of this article the term 'user' does not necessarily mean an end-user. For example, if you're writing a software library your *user* is the guy who uses your library. The same guy is usually a developer of another product and is therefore a supplier for another developer or for an end-user. This kind of multi-tier supplier-consumer relations is nothing new, and goes back at least for a thousand years, to the time when the carpenter acted both as producer of a house for the end-user and as a consumer of nails produced by the blacksmith.

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams [Adams].

Sergey Ignatchenko has 12+ years of industry experience, and recently has started an uphill battle against common wisdoms in programming and project management. He can be contacted at si@bluewhalesoftware.com

Relevance of business requirements

In traditional (non-agile) development models users rarely interact with developers directly. In non-agile teams, as well is in many agile ones, the tasks usually come to developers (or business analysts) in the form of business requirements. Unfortunately, way too often these requirements are not clear enough. But even worse, often there are requirements which are not really relevant to keeping users happy. In such cases the impact on development can easily be devastating – if developers are forced to do something outright stupid, one cannot possibly expect them to work with enthusiasm.

The big question here is how to distinguish relevant business requirements from irrelevant ones? The answer is quite straightforward: whatever is related to keeping users happy is potentially relevant. Applying this principle to practical situations can lead to not so trivial results, so let's consider a few examples. Let's consider a situation when an application is being developered for a mobile phone. One potentially valid business requirement in this case is 'our application should run on an iPhone', and if developers are trying to fight it (on any grounds) they're most likely out of their depths. It is worth noting that this requirement should be specified exactly as 'our application should run on an iPhone', and not as 'our application should use iOS' – even if using iOS will eventually turn out to be the only way to run the application on an iPhone, it is an

1. Quote by Garfield the Cat from [Garfield88]

Hey, you're talking about the **importance of the end-user**, but the end-user **clearly wants something 'cool'**

'implementation detail', and therefore a decision which should be made at the architectural level rather than at the business level. As an alternative example, if the product is a software library then the requirement 'it should be portable to iOS' is a perfectly valid one – in this case the OS requirement becomes a characteristic which can be observed by the product user.

It's so 1990-ish

One issue which often emerges within development teams is the question: 'Hey, why don't we use this new cool technology? C++ is so 1990-ish!'. My usual answer (perfectly consistent with the logic I've described above) is that 'cool' doesn't have any standing in my books and that we should think about the user first, and that with this new cool technology user will suffer in this or that way. Usually this kind of explanation about overall project success and being user-oriented does help, but recently I've run into a counter-argument: 'Hey, you're talking about the importance of the end-user, but the end-user clearly wants something 'cool', look at the iPhone and iPad! So why don't you allow us to use cool stuff?!'.

While this logic is still flawed, to illustrate why will need a bit more of an explanation. When users use the word 'cool' they're completely within their rights to ask for whatever they want and developers should listen to them. In other words, within 'userland' (a.k.a. 'managerland' and 'marketingland' – and don't confuse it with *nix 'userland') the word 'cool' is a perfectly legitimate argument, and hence a valid business requirement and developers must learn to live with it. But when developers starts to use word 'cool' to describe technology which their users do not care about, it has nothing to do with users and therefore should have much less priority if considered. There are two completely separate worlds: one is 'userland', the other is 'developerland', and 'cool' only has standing within 'userland'. While it may seem 'unfair' to developers it is a direct result of the asymmetry described above and users having the upper hand.

Developers and user interfaces

Another area of everlasting conflict between users and developers are user interfaces, with many a fight over usability. One of my fellow-rabbits even uses the special term 'developer's UI' to describe one which was convenient to write but is hardly usable. The worst example I've personally seen to date was a certain fax machine (I will not name the company here, but anybody who's seen it should recognize it easily). It was a nightmare UI to deal with, despite having all the necessary features. For example, after the fax has been sent it showed the notification 'N pages sent ok', but why did this disappear after a few seconds? Did they expect me to be right next to the machine all the time to catch a glimpse of it? Or why, if the sending had failed did it go into one of two different, but visually very similar, modes – one with a retry being scheduled and another with the whole thing aborted? And in order to cancel the retry, why did you need to go three levels deep into the menus, under the heading 'memory settings'? I am a developer myself and I perfectly understand why it was written this way – for a developer it is so much easier to design a UI around the *implementation* (or even worse – around an unsuitable *existing implementation*), but as a user I clearly have difficulties with finding non-foul words to describe the experience; needless to say, chances of me buying another fax machine from the same company are on the order of me voluntary paying a visit to a pre-heated farmer's oven.

While technically speaking it is not a job of a software developer to design a UI (ideally, the task should belong to business analysts), whenever a developer (who wants the project to succeed) is implementing a UI (whether inventing it him/herself, or implementing specification), s/he should think about the user who will use the product. While it doesn't help 100% of the time – an average user can have expectations which are very different from an average developer – it still can help to avoid at least the

the developer tends to **concentrate on the areas** which he thinks are of interest from the **point of view of implementation**

most blatant problems. Just don't forget to discuss it with the business analyst before deviating from the existing specification – it might be good not only to save you some trouble, but also sometimes can be useful for the project and end-user too.

Eating our own rabbit food

It should be mentioned that it is often difficult to think about your own code from the point of view of a user, especially when it is already written. In this case it becomes very similar to testing your own code, which is known for fellow-rabbits to be very difficult. One reason for this difficulty is that such testing puts you into position of perceived conflict of interest: if you find the bug or other flaw (which is your job as a tester), it means that you have made a mistake as a developer. While this conflict of interest is usually only perceived and is not a real one, it often still leads to situations when the developer/tester subconsciously avoids testing scenarios which can be dangerous. Another (probably even bigger) problem in this way is that during such testing the developer tends to concentrate on the areas which he thinks are of interest from the point of view of implementation; while such 'white-box testing' is indeed useful, it tends to differ from the usage patterns of users.

One obvious way to deal with these issues is to have an independent QA department; another technique which helps is known as 'eating your own rabbit food' (or 'eating your own dog food' among some lesser species). This means that the company should use its own products as much as possible, to experience them as a user. While this technique alone does not provide any guarantees, it certainly can be a good tool to improve the overall user experience.

The manager's perspective (team-leads included)

In this 'user vs developer' conflict, managers find that being between the user and the developer is very similar to being between a hammer and an anvil. It applies to all levels of the management, from the top level down to team leads. From one side there is a pressure to make a product successful (and to achieve that by making users happy), from the other side an obvious lack of understanding (and therefore inertia, if not outright opposition) from the developers. It is indeed a difficult problem for management, but it can be solved (as described, for example, as early as in [Parkinson60]) by promoting a culture where everybody works towards a well-defined goal – project success (and therefore making user happy). How to achieve this is not a trivial management task (it goes much further than simple stock options and other incentives), but it is certainly do-able. One notorious example of succeeding at this is Louis Gerstner's highly successful restructuring of IBM in the 1990s; while re-establishing a customer-oriented culture obviously wasn't the only change which led to this success, this cultural shift certainly was a significant part of Gerstner's plan. As several fellow-rabbits who had a chance to work in IBM have told me, it was Gerstner who allowed IBM integrators to use non-IBM solutions when it was necessary to make customers happy. And as we can see 10 years down the road, it was a highly successful strategy.

The developer's perspective

One question some developers ask – 'ok, you have shown that project success depends on the user, but why I should care?'. Unfortunately (consistent with [Parkinson60]), there is no good answer to this question, except that organization where nobody cares about results is inevitably doomed. If all you want in this life is to be able to pay your bills, and caring about results (and therefore about the user) is not strictly necessary. Still, as the experience of the whole rabbit community shows, projects which are successful have a much higher chance of being kept even during a crisis, and to provide higher raises when the economy is booming, so thinking about user often pays off even in a direct monetary sense.

Going a bit further with this analysis: if you're working for a company (department, project, etc.) where management and developers don't care about the eventual success of what they're doing, it often means that the company is likely to fail. Working for a company which is doomed to failure is never a good thing. It is bad for your personal bottom line, not really helpful for your career, and can be devastating for your self-esteem. In short – a developer who can do better than fail should aim to avoid such workplaces, and try to get into an environment where the culture of project success is predominant on all the levels. It will certainly require more effort, but has much more potential to be much more rewarding, both financially and emotionally. ■

References

[Adams] http://en.wikipedia.org/wiki/Lapine_language

[Garfield88] *Garfield and Friends*, 1st season, CBS, 1988

[Loganberry] David 'Loganberry', *Frithaes! - an Introduction to Colloquial Lapine!*, http://www.scribd.com/doc/97067/Conlang-Lapine

[Parkinson60] Cyril Northcote Parkinson, *The Law and the Profits*, 1960, ASIN B004NDFID4

Exception Specifications in C++ 2011

The new standard is almost finished. Dietmar Kühl looks at the new exception features.

This article discusses exception specifications in C++ 2011. The primary focus of this discussion is the new **`noexcept`** keyword and the related concepts. The reason for this strong focus is simply that exception specifications for anything else than the distinction between functions which may throw and functions which will not throw are a failed experiment. This will discussed at the end of this article. However, the specification that a function will not cause any exception is an important piece of information.

Motivation

It is just fun to watch a bunch of experts discuss something which is supposedly entirely under control, just to find shortly prior to shipping a product that just a tiny detail got missed. Taking advantage of move construction to optimize libraries is one such topic which got discussed for ages, always essentially under the assumption that move construction won't throw any exceptions. Well, eventually it transpired that there are examples where moving an object might throw. This is the tiny detail which got missed. Once people looked more closely at the issue it turned out that this slight oversight actually turned into a tremendous monster, threatening to make use of move construction in the standard library impossible at all: the typical type currently in existence doesn't have a move constructor. Instead, moving an object would actually turn into using its copy constructor and the assumption that copy constructors don't throw exceptions is adventurous at best. The only positive aspect of this mess is that it actually was caught prior to shipping!

Let's start with an example of the problem: assume we want to implement the class template **`std::vector<T>`** (ignoring the allocator because this particular issue doesn't even need an allocator to land itself in a mess). To be more concrete, we want to implement this class's **`reserve()`** method: this method increases the number of available elements before the internal memory needs to be rearranged. The particular aspect of its specification we are interested in is that this function leaves the **`std::vector<T>`** unchanged if an exception occurs. The C++2003 approach to implement the case where additional memory needs to be allocated (**`reserve()`** is a bit more complex but we aren't really interested in the other cases for this discussion) is to do the following:

1. allocate enough memory to hold the requested number of elements
2. copy the elements into the newly allocated memory
3. set up **`std::vector<T>`**'s data structure to use the new storage
4. destroy the original sequence of elements
5. deallocate the original memory.

Note that this actually only works correctly if neither the destructor of the element type **`T`** nor the deallocation function ever throw an exception. This is already a requirement for the type **`T`** in C++2003. This implementation provides the strong exception guarantee (i.e. the **`vector<T>`** stays unchanged if an exception is thrown during **`reserve()`**):

- If an exception is thrown during step 1 the object isn't changed, yet.
- If an exception is thrown during step 2 the original sequence isn't changed, yet, either; for proper clean up the already copied elements need to be destroyed and the allocated memory needs to be released but this is doable quite easily as none of those clean-up operations may throw an exception.
- After this, the operations are only manipulations of built-in types which don't throw, destructor calls, and releasing the memory, none of which is allowed to throw an exception.

Of course, in C++2011 we can do better: we can move the objects rather than copying them! That is, we replace step 2 to just move the elements (at first sight it seems as if we could get rid of step 4, too, but even after moving the objects still exist and they need to be destroyed). The problem is that in general moving an object may throw! If this happens, we need to restore the original state of the sequence which would involve moving the objects back. Of course, if a move just threw, we have no guarantee that moving an object back doesn't throw, too! That is, once a move threw, we can't necessarily restore the original sequence. If we can't restore the original sequence we can't use **`std::move()`** (or whatever other approach we use to move objects) to implement **`std::vector<T>:reserve()`**.

This same logic essentiallly applies to many other places where moving an object would be beneficial: if the move operation may throw and thus cannot be undone safely it cannot be used in most places where the strong exception guarantee is given. As a result moving objects would be restricted to a few places in the standard C++ library. This is essentially an unacceptable prospect.

To prevent C++ standard melt-down people went ahead and constructed a solution for the problem which, unfortunately, is somewhat painful due to some of the choices made. At least, it seems to work: functions can be declared not to throw any exception using either the now deprecated empty throw specification – **`throw()`** – or the newly introduced **`noexcept`** specification. A **`noexcept`** specification can take a constant expression evaluating to **`bool`** making it conditional. And **`noexcept`** can be used as an operator, determining whether an expression can throw an exception at compile-time i.e. the **`noexcept`** operator is a constant expression.

The key idea is that the implementer of a function knows in many cases that the function won't throw an exception. For example, often **`move`** construction of a resource-owning class just moves pointers around and sets the original pointers to **`null`**. None of these operations throws. That is, in many cases moving the objects is viable but it is necessary to tell the system that the move constructor doesn't throw. Operations capable of possibly moving objects rather than copying them then use a **`noexcept`** operator to detect whether the move is guaranteed not to throw.

Dietmar Kühl is a senior developer at Bloomberg L.P. working primarily on energy and emissions related models. He is also a frequent attendee of the C++ standardization meetings and one of the moderators of comp.lang.c++.moderated. He can be reached at dkuhl@bloomberg.net.

existing code is lacking proper exception specifications and it is often not an option to change some of the code

In principle, an empty throw specification in C++ 2003 pretty much says the same thing as a **`noexcept(true)`** specification or an empty throw specification in C++ 2011. In both cases an exception leaking out of the corresponding function causes the program to **`std::terminate()`**. However, there are a number of important details:

1. If an exception is leaking from a function with a **`throw()`** specification (both in C++ 2003 and C++ 2011), the program is terminated following a call to **`std::unexpected()`** (in the general case **`unexpected()`** could change the exception into one which is expected but with an empty throw specification there is no exception which would be expected). However, before **`std::unexpected()`** is called the stack is unwound up to the function with the **`throw()`** specification. In the case of an exception escaping from a function declared **`noexcept(true)`**, stack unwinding may or may not happen as the system sees fit and **`std::terminate()`** is called directly without also calling **`std::unexpected()`**. This difference allows for some optimizations which are thought to remove any potential overhead from functions declared not to throw anything. The empty throw specification may have a performance impact even if no exception ever escapes the corresponding function.
2. It isn't possible to have a conditional empty throw specification but the **`noexcept(expr)`** specification takes an optional Boolean constant expression as argument, i.e. the **`noexcept`** specification can be conditional: if the passed expression **`expr`** evaluates to **`false`** a function can throw exceptions; otherwise, the function is not allowed to throw any exception. This is especially important for function templates: depending on the template parameters expressions may or may not throw exceptions. With a conditional specification it is possible to take advantage of this.
3. In C++ 2003 it cannot be determined at compile time if a function is declared with an empty throw specification. This is changed in C++ 2011 where it can be determined whether a function is not allowed to throw any exception i.e. whether it has an empty throw specification or a **`noexcept(true)`** specification: the expression **`noexcept(expr)`** (note that **`noexcept`** is used as an operator here) yields a Boolean compile-time constant indicating if all operations in the expression **`expr`** are declared to be **`noexcept(true)`** or **`throw()`**.

Originally, the **`noexcept`** specification was intended to be much stronger: it would be a compile-time error trying to use any operation which isn't **`noexcept(true)`**. This would have meant that calling any function without a **`noexcept(true)`** specification would have to be wrapped into a try/catch block where one of the catch blocks would have been required to catch any exception using (**`...`**). This very strong requirement was dropped because it was considered to hamper transition to **`noexcept(true)`** use. The use of a **`throw()`** specification is discouraged because it may incur some run-time overhead even if no exception is thrown. In C++ 2003 this is the only way to specify that no exception will be thrown. Thus, existing code is lacking proper exception specifications and it is often not an option to change some of the code. Adding **`noexcept(true)`** specifications to code based on functions lacking proper exception specifications would either fail to compile or require try/catch wrappers if it were an error to call a function which isn't **`noexcept(true)`**.

The flipside of the coin is that now it becomes necessary to protect functions using **`noexcept(true)`** specifications against changes in underlying libraries who are specified to not throw exceptions originally: while this guarantee exists when the **`noexcept(true)`** specification was added nothing prevents removal of the corresponding exception specifications. This silent change would potentially cause programs to abort because an optimization was safe at some point but was broken because some lower-level code got changed. Although changing a **`noexcept(true)`** specification to a **`noexcept(false)`** specification effectively amounts to an interface change, it is a change which isn't detected by the compiler. Having to use either conditional **`noexcept`** specifications or **`static_assert()`**s to prevent code from silently breaking, at least if it uses user-defined functions, is quite painful. Static analysis may yield warnings about potentially throwing operations and I'd think this would be a valuable tool given that any accidentally thrown exception terminates the program. I would have preferred an error or, if this is deemed not acceptable as a default, some way to opt into stronger checks (e.g. a **`noexcept`** block which causes an error if potentially throwing operations are used within).

noexcept-specification

Functions can be declared to never throw any exception by using a **`noexcept`** specification following the function declaration, e.g.:

```
void f() noexcept;
```

This declaration could alternatively have used an empty throw specification:

```
void f() throw();
```

These two declarations of **`f()`** are compatible: both declarations can appear in a program and with respect to the declaration their meaning is identical: both declare that **`f()`** can't throw an exception. Incompatible exception specifications, i.e. exception specifications allowing/ disallowing different exceptions are not permitted. However, the definition of the function determines how the program proceeds if an exception escapes the function with the exception specification: if the function definition uses a **`noexcept`** specification, **`std::terminate()`** is called immediately; if the function definition uses a dynamic exception specification, i.e. **`throw()`**, then **`std::unexpected()`** is called which will ultimately call **`std::terminate()`**, too. The key difference between both approaches is that before calling **`std::unexpected()`** the stack is unwound while there is no such guarantee when **`std::terminate()`** is called immediately.

a destructor without an exception specification which throws an exception gets an exception specification which may very well disallow any exceptions

The **noexcept** specification can take a Boolean constant expression as argument which determines whether the function should be **noexcept**. For example:

```
void g() noexcept(true);
// same as noexcept without argument
void h() noexcept(false);
template <typename T>
   void m() noexcept(noexcept(T()));
```

The first two examples are rather simple: **g()** is declared to never throw any exception and **h()** is declared to possibly throw an exception. Whether the function **m()** is allowed to throw an exception depends on the template parameter **T**: using a **noexcept** operator (see below) this specification determines whether the default constructor or the destructor of **T** might throw an exception. If they don't, the function **m()** is specified not to throw any exception, either. Otherwise, i.e. if the default constructor or the destructor of **T** might throw an exception, **m()** might throw an exception as well. Note that the expression **T()** constructs a temporary object using the default constructor and destroys the object again. Thus, both the default constructor and the destructor are considered!

With explicit exception specifications the situation is rather straightforward: if there is an exception specification, a function just gets the corresponding specification. It gets more interesting if there is no exception specification. In this case a function is normally allowed to throw exceptions. Most of the time this is what is desired but it would also cause all destructors without exception specification to be considered throwing although it has long been recommended that destructors should be non-throwing. In addition, it is very desirable that destructors can be detected as being non-throwing. Therefore, destructors get special treatment: if the destructor has no explicit exception specification, it gets the same exception specification an implicitly generated destructor would get. So, let's see what happens there.

There are a number of implicitly generated functions for which it would be desirable to have **noexcept(true)** specifications where possible. This applies to the default constructor, copy constructor, move constructor, destructor, copy assignment, and move assignment, commonly called the *special functions*. When one of these special functions is implicitly generated, it gets an exception specification which allows all exceptions of all operations called by the implicitly generated special function. For example, the implicitly generated copy constructor copies all bases and all members. The corresponding exception specification will consist of a union of all exceptions specified by the copy constructors of the bases and the members. This exception specification is possibly equivalent to **noexcept(false)**, i.e. all exceptions are allowed: this is the case if at least one of the base or member copy constructors allows all exceptions. If none of the copy constructors of the members or bases can throw any exception, i.e. they are all declared to be **noexcept(true)**, then the exception specification of the generated copy constructor will also be **noexcept(true)**.

Now, for destructors these rules apply even if the destructor is explicit but has no exception specification. That is, if your destructor does throw an exception but none of the destructors of the bases or members throws exceptions, you will have to declare explicitly that the destructor might throw an exception (of course, you should consider whether your destructor really needs to throw as well). To allow an explicitly implemented destructor to throw an exception it needs to either have a **throw(exceptions)** specifier or it needs to get a **noexcept(false)** specification. Somehow this latter declaration reminds me of 'Yes! We have no bananas'!

```
typedef int the;
int const up(42);

struct the_oracle
{
    ~the_oracle() { throw up; }
};

int main()
{
    try { the_oracle(); }
    catch (the answer) {}
}
```

Listing 1

Note that this is a silent change: a destructor without an exception specification which throws an exception gets an exception specification which may very well disallow any exceptions! I couldn't verify this behaviour on any of the compilers I have currently available, however. This may be due to this being a relatively recent change (it made its first appearance in N3225). Listing 1 is the example program I used to test.

The destructor of **the_oracle** will acquire a **throw()** or a **noexcept(true)** exception specification (which one is unclear in the standard; the difference is whether the stack is partially unwound or not, respectively, before the program is terminated). Either will cause the program to be terminated instead of returning normally from **main()**. Yes, I know that it is bad practice to throw from a destructor but this doesn't mean that there is no code out there which benefits from having this change pointed out.

noexcept operator

To determine whether an expression might, according to its exception specifications, throw an exception, the **noexcept** operator can be used:

```
template <typename T>
void f() {
  if (noexcept(T()))
    ...
```

In this example the **noexcept** operator tests if the default constructor and the destructor for the template argument **T** are specified not to throw any

freedom to strengthen the exception specifications isn't given for virtual functions

exception: if the expression `T()` were executed it would use the default constructor to create a temporary which would then be destroyed using the destructor. Thus, both operations have to be declared not to throw an exception for the `noexcept` operator to yield `true`. The result of the `noexcept` operator is a constant expression and the expression passed to the `noexcept` operator is not evaluated i.e. if the expression has side effects, these won't happen (just the same as `sizeof` which is also a constant expression with the argument not being evaluated).

In general, the argument to the `noexcept` operator can be an arbitrary expression. The `noexcept` operator yields `false` if there is any potentially evaluated expression or subexpression which may throw an exception, i.e. if there is any expression or subexpression which is not specified to be `noexcept(true)` or `throw()`. Note that this includes any implicit operations needed by the expression like implicit conversions or destructors. Of course, any `dynamic_cast<>()` on references, throw expression, or `typeid` expression would also cause the `noexcept` operator to yield `false`. However, if all expressions or subexpressions are specified to be `noexcept(true)` or `throw()` the `noexcept` operator yields `true` as well.

The `noexcept` operator is the key using non-throwing operations in optimizations for any sort of templatized component. Let's get back to the original example of implementing `std::vector<T>::reserve()`: this function could determine whether objects can be moved without ever throwing an exception using an expression like this:

```
  noexcept(T(std::declval<T>()))
```

The function template `std::declval()` is declared to have a return type matching its template argument, possibly turned into an r-value, and is specified not to throw an exception itself. It is only declared and has no definition, however:

```
  template <typename T>
  typename std::add_rvalue_reference<T>::
     type declval() noexcept;
```

With this, the expression above just tests whether it is possible to both construct an object of type `T` from a movable `T` object and to destroy an object of type `T` without throwing any exception. Although the `noexcept` operators can be used, it is worth pointing out that the header `<type_traits>` defines a number of type traits which do this test more directly. Likewise, there is a function declared in the header `<utility>` which takes care of all the needs for moving vs. copying of objects: `std::move_if_noexcept()` is similar to `std::move()` but the result type is only an r-value reference if the `move` constructor of the argument type is specified to be `noexcept(true)`.

Although the `noexcept` operator is an important facility in the C++ tool box, it is similar to the `sizeof` operator: the `noexcept` operator is probably rarely used explicitly. However, type traits and auxiliary functions effectively based on this operator are likely to show up in many places: it is beneficial to expose that operations won't throw any exception in many places and libraries trying to be as efficient as possible will make use of this knowledge.

User and standard library use

In general, it seems as if `noexcept(true)` specifications should be freely used wherever it is known that a function can't throw an exception. Given that an incorrect `noexcept(true)` specification might terminate the program it should not be applied carelessly, however. In practice this probably means that many functions which could be specified to be `noexcept(true)` won't get this specification. Hopefully, the danger of wrongly using `noexcept(true)` will be mitigated by compilers and/or static analysers which could warn about dangerous `noexcept(true)` specifications or suggest functions which could safely be made `noexcept(true)`. Given that this is a brand-new feature I don't expect any such tool to be around already. Also, it is probably worth verifying that `noexcept(true)` specifications have indeed no adverse run-time effect.

From a semantic view the `noexcept(true)` specification should have no impact. It is worth noting, however, that declaring a move constructor to be `noexcept(true)` will cause several standard library algorithms to use the move constructor rather than the copy constructor. Obviously, these constructors should be implemented in a way which makes it viable to use a move constructor instead of a copy constructor. For implicitly generated constructors this is already the case, assuming that any user-defined constructors used to deal with subobjects have this property, too. In practice, specifying a `move` constructor to be `noexcept(true)` should just make the program faster.

Although `noexcept(true)` can be useful for many functions its use for now, at least, mainly affects algorithms in the standard library. These mainly care about some of the special functions and the `swap()` function: all of the operations which might be involved in moving objects, i.e. the move constructor, the move assignment, the destructor, and the `swap()` function should, whenever possible, be written not to throw any exceptions. In most cases these operations are probably simple pointer operations which can't throw an exception. Any of these operations which indeed never throws an exception should also be declared to be `noexept(true)`. This is the area where the biggest performance gains from moving over copying are expected.

In the standard library there are certain functions which are required to be `noexcept(true)`. Some functions have to be specified conditionally `noexcept(true)`. For the vast majority of functions in the standard library the standard makes no requirement that the function has to be specified `noexcept(true)`. However, the library implementer is free to specify any non-virtual function for which it is known that no exception is thrown `noexcept(true)`. It is expected that the library implementers make use of this freedom. The freedom to strengthen the exception specifications isn't given for virtual functions: since the overriding functions have to apply the same exception specification or stronger than the base class version, allowing the library implementer to strengthen the exception specification for any virtual function would cause incompatible implementations.

Use in the library specification

The original approach for `noexcept` specifications in the standard C++ library specification was to make any function which has a clause saying 'throws nothing' `noexcept(true)`. This seems like a sensible approach but actually it is not! The reason for this is not necessarily obvious: a lot of the functions have preconditions and if these preconditions are met, the function indeed doesn't throw any exception. However, what happens when a precondition is not met? The answer is that this causes undefined behaviour, i.e. anything, including throwing an exception, could happen. For example, a checking version of the standard library might detect that a precondition isn't met and signal this by throwing an exception. If the corresponding function were required to be `noexcept(true)` this safe implementation of the standard library would cause a rather unhelpful termination. Thus, the rules when to require `noexcept(true)` got revised to cater for this.

To better describe the rules used to determine whether a function is made `noexcept(true)` or not, it is helpful to define two terms:

- A function without any precondition is said to have a **wide contract**. Obviously, for member functions the object on which such a function is called actually does have the precondition that the object exists. An example of a function with a wide contract is `std::vector<T>::size()`: whenever there is corresponding vector object around, this function can be called
- A function which has some precondition is said to have a **narrow contract**. `std::vector<T>::pop_back()` is an example of a function with a narrow contract because it requires that there is at least one element in the vector. However, its specification still says that it throws nothing.

The rules for when `noexcept(true)` is to be applied for standard library functions which are specified to throw nothing depend on whether the respective function has a wide or a narrow contract: since functions with wide contracts cannot be abused, i.e. there is no precondition which can be violated, it is safe to make them `noexcept(true)` if they can't throw an exception. However, any function which has a narrow contract, even though it may be specified to not throw any exception when it is correctly used, can throw an exception if the precondition is violated. That is, no function with a narrow contract is required to be `noexcept(true)`.

An implementation which doesn't do any checking of the precondition is allowed to strengthen the exception specification, however. This means that the corresponding function may actually get different throw specifications in different build configurations: in a safe mode where the preconditions are checked and violations are signaled via an exception, the function is `noexcept(false)` while in a release build where the preconditions are not checked it may become `noexcept(true)`.

Exception specifications in general

While the distinction between throwing functions and non-throwing functions is rather useful, especially if this property of a function can be detected, more general exception specifications are not. Essentially, the idea to declare what kind of exceptions a function can throw is a failed experiment. Well, if someone had thought hard enough about this it wouldn't even have been necessary to run an experiment! The intention of exceptions is to relieve business logic from forwarding error information. The goal is to allow handling of exceptional errors at an appropriate level without interfering with the business logic. Obviously, adding a specification of what errors might happen in a specific function is nothing else than burdening the business logic with information on error forwarding – something directly defeating the goal of exceptions. Yes, it isn't in the body of the code and to some extent separated but it is still there and typically has nothing to do with the business logic other than exposing some more or less random implementation details.

I'd think this is already bad enough but it actually becomes worse! Most interesting functions are generic in some form, be it that they are virtual, take a template argument or a function pointer, or depend on other objects which are somehow customizable. How on earth can an author of such a function even dream of possibly anticipating what kind of exception will be thrown in the setup in which I am calling it? Any exception specification just becomes an unnecessary road-block which would require me to disguise my carefully crafted exception hierarchy as some amorphous piece of junk which later needs to be recovered carefully and inspected at every potential catch-site.

With C++ 2011 the class `std::exception_ptr` and the related means to get hold of such a beast give me the opportunity to disguise my exceptions in a reasonably generic way (e.g. I could construct some sort of chain of disguised exceptions) which wasn't available earlier. Note, however, that this is not the intention of this class! Instead, it is intended to marshal exceptions unchanged from one thread of execution to another one. Yes, it could be used to create chains of disguised exceptions so that we can play nicely with exception specifications. Of course, having to remember that the exception which was caught needs to be demarshaled and investigated for its actual content makes the processing of exceptions harder rather than easier. Also, I have actually lost the claimed benefit of exception specifications because I actually need to handle errors which are explicitly not specified. The use of a tool which makes life harder in return for no benefit whatsoever is hardly advisable.

Just to be explicit for those who wonder: this assessment is actually not specific to C++! It applies to all languages which are misled to participate in this experiment. Maybe the actual experiment is to see how many programming language communities can be made to believe that a specification of which exceptions a function might throw is somehow a good thing? Obviously, this isn't related to the `noexcept` discussion but I needed to get this off my chest anyway.

Conclusions

With C++2011 it becomes possible to conditionally specify functions to not throw any exceptions. In addition, it becomes possible to detect whether any given expression might throw an exception. The positive end of this is that a lot of operations may become faster by taking advantage of the fact that there are no exceptions. At the negative end, getting the exceptions specification wrong will `std::terminate()` the program and the user has to manually add the exception specifications to all but the implicitly generated special functions which deduce their exception specification from the operations they implicitly call. Hopefully, there will soon be tools which warn about `noexcept(true)` specifications in functions which actually might throw or suggest adding `noexcept(true)` for functions which are known not to throw but don't have the corresponding exception specification, yet. ■

Acknowledgement and references

Thanks to Daniel Krügler, Jens Maurer, and Nicolai Josuttis for their very constructive feedback on several drafts of this document.

This article is based on several documents used during the C++ standardization:

N2855 – Rvalue References and Exception Safety (Douglas Gregor, David Abrahams)

N3225 – Working Draft, Standard for Programming Language C++ (Pete Becker) as of 2010-11

N3291 – Working Draft, Standard for Programming Language C++ (Pete Becker) as of 2011-04

N3279 – Conservative use of noexcept in the Library (Alisdair Meredith, John Lakos)

Why Interval Arithmetic Won't Cure Your Floating Point Blues

We've looked at several approaches to numeric computing. Richard Harris has a final attempt to find an accurate solution.

In this series of articles we have discussed floating point arithmetic, fixed point arithmetic, rational arithmetic and computer algebra and have found that, with the notable exception of the unfortunately impossible to achieve infinite precision that the last of these seems to promise, they all suffer from the problems that floating point arithmetic is so often criticised for.

No matter what number representation we use, we shall in general have to think carefully about how we use them.

We must design our algorithms so that we keep approximation errors in check, only allowing them to grow as quickly as is absolutely unavoidable.

We must ensure that these algorithms can, as far as is possible, identify their own modes of failure and issue errors if and when they arise.

Finally, we must use them cautiously and with full awareness of their numerical properties.

Now, all of this sounds like hard work, so it would be nice if we could get the computer to do the analysis for us.

Whilst it would be extremely difficult to automate the design of stable numerical algorithms, there is a numeric type that can keep track of the errors as they accumulate.

Interval arithmetic

Recall that, assuming we are rounding to nearest, the basic floating point arithmetic operations are guaranteed to introduce a proportional error of no greater than 1±½ε.

It's not wholly unreasonable to assume that more complex floating point functions built into our chips and libraries will also do no worse than 1±½ε.

This suggests that it might be possible to represent a number with upper and lower bounds of accuracy and propagate the growth of these bounds through arithmetic operations.

To capture the upper and lower bound of the result of a calculation accurately we need to perform it twice; once rounding towards minus infinity and once rounding towards plus infinity.

Unfortunately, the C++ standard provides no facility for manipulating the rounding mode.

We shall, instead, propagate proportional errors at a rate of 1±1½ε since we shall be able to do this without switching the IEEE rounding mode. Specifically, we can multiply a floating point result by 1+ε to get an upper bound and multiply it by 1-ε to get a lower bound. The rate of error propagation results from widening the 1±½ε proportional error in the result by this further 1±ε to yield the bounds.

Naturally, this will not work correctly for denormalised numbers since they have a 0 rather than a 1 before the decimal point, so we shall have to treat them as a special case. Specifically, rather than multiplying by 1±ε, we shall add and subtract the smallest normal floating point number, which is provided by **`std::numeric_limits<double>::min()`**, multiplied by ε.

Before we apply rounding error to the upper and lower bounds of the result a calculation we shall want them to represent the largest and the smallest possible result respectively. We therefore define the basic arithmetic operations as follows.

$$(a, b) + (c, d) = (a + c, b + d)$$

$$(a, b) - (c, d) = (a - d, b - c)$$

$$(a, b) \times (c, d) = (\min(a \times c, a \times d, b \times c, b \times d), \max(a \times c, a \times d, b \times c, b \times d))$$

$$(a, b) \div (c, d) = (\min(a \div c, a \div d, b \div c, b \div d), \max(a \div c, a \div d, b \div c, b \div d))$$

As an example, consider the square root of 2. Working with 3 accurate decimal digits of precision, this would be represented by the interval

$$(1.413, 1.415)$$

Squaring this result would yield

$$(1.413 \times 1.413 - 0.001, 1.415 \times 1.415 + 0.001)$$

or

$$(1.996, 2.003)$$

which clearly straddles the exact result of 2 and furthermore gives a good indication of the numerical error.

Unfortunately, as is often the case, the devil is in the details.

The first thing we need to decide is whether the bounds are open or closed; whether the value represented is strictly between the bounds or can be equal to them. If we choose the former we cannot represent those numbers which are exact in floating point, like zero for example, so we shall choose the latter.

However, if either of the bounds is infinite we should prefer to consider them as open so as to simplify the rules of our interval arithmetic. This means that we must consequently map the intervals (-∞, -∞) and (∞, ∞) to (NaN, NaN).

A consequence of treating infinite bounds as open is that multiplying them by zero can yield zero rather than NaN. In particular, we have a special case of

$$(-\infty, \infty) \times (0, 0) = (0, 0)$$

and hence

$$\frac{(0, 0)}{(0, 0)} = (-\infty, \infty)$$

This might seem a bit odd, but it makes perfect sense once we realise that (-∞, ∞) is pronounced 'any number'.

Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

we can consistently, albeit not particularly mathematically soundly, define division by intervals containing zero or either infinity

Making certain assumptions we can consistently, albeit not particularly mathematically soundly, define division by intervals containing zero or either infinity as follows. Given

$$0 < a \le b$$
$$0 < c \le d$$

we define

$$(-\infty,\infty) = \frac{(a,b)}{(0,0)} = \frac{(-a,b)}{(0,0)} = \frac{(-b,-a)}{(0,0)} = \frac{(0,0)}{(0,0)}$$
$$(-\infty,\infty) = \frac{(a,b)}{(-c,d)} = \frac{(-a,b)}{(-c,d)} = \frac{(-b,-a)}{(-c,d)} = \frac{(0,0)}{(-c,d)}$$
$$(-\infty,\infty) = \frac{(-a,b)}{(0,d)} = \frac{(-a,b)}{(-c,0)}$$
$$(0,\infty) = \frac{(a,b)}{(0,d)} = \frac{(-b,-a)}{(-c,0)} = \frac{(0,b)}{(0,d)} = \frac{(-a,0)}{(-c,0)}$$
$$(-\infty,0) = \frac{(a,b)}{(-c,0)} = \frac{(-b,-a)}{(0,d)} = \frac{(0,b)}{(-c,0)} = \frac{(-a,0)}{(0,d)}$$

and

$$(0,\infty) = \frac{(a,\infty)}{(c,\infty)} = \frac{(-\infty,-a)}{(-\infty,-c)}$$
$$(-\infty,0) = \frac{(a,\infty)}{(-\infty,-c)} = \frac{(-\infty,-a)}{(c,\infty)}$$

```
class interval
{
public:
  static void add_error(double &lb, double &ub);

  interval();
  interval(const double x);
  interval(const double lb, const double ub);

  double     lower_bound() const;
  double     upper_bound() const;
  bool       is_nan() const;
  int        compare(const interval &x) const;
  interval & negate();

  interval & operator+=(const interval &x);
  interval & operator-=(const interval &x);
  interval & operator*=(const interval &x);
  interval & operator/=(const interval &x);

private:
  double lb_;
  double ub_;
};
```

Listing 1

An interval class

Listing 1 gives the definition of an interval number class.

In the constructors we shall assume that a value is inexact, unless the user explicitly passes equal upper and lower bounds or it is default constructed as 0. Note that the user can also capture measurement uncertainty with the upper and lower bound constructor.

The **`add_error`** static member function is provided to perform the error accumulation for both member and free functions. Its definition is provided in listing 2.

For the upper bound, this worst case will occur if we multiply by **`u`** when it is positive and by **`l`** when it is negative. This is because, for negative numbers, the greater numbers are those with smaller magnitudes. By the same argument the opposite is true for the lower bound.

Additionally we must take care to accumulate errors in denormal numbers additively.

Finally we exploit the fact that all comparisons involving NaNs are false to ensure that if either bound is NaN, both will be.

```
void
interval::add_error(double &lb, double &ub)
{
  static const double i
     = std::numeric_limits<double>::infinity();
  static const double e
     = std::numeric_limits<double>::epsilon();
  static const double m
     = std::numeric_limits<double>::min();
  static const double l = 1.0-e;
  static const double u = 1.0+e;

  if(lb==lb && ub==ub && (lb!=ub
     || (lb!=i && lb!=-i)))
  {
    if(lb>ub)  std::swap(lb, ub);

    if(lb>m)        lb *= l;
    else if(lb<-m)  lb *= u;
    else            lb -= e*m;

    if(ub>m)        ub *= u;
    else if(ub<-m)  ub *= l;
    else            ub += e*m;
  }
  else
  {
    lb = std::numeric_limits<double>::quiet_NaN();
    ub = std::numeric_limits<double>::quiet_NaN();
  }
}
```

Listing 2

Negating an interval can be done without introducing any further error since it simply requires flipping the sign bit

The implementations of the constructors, together with the trivial upper and lower bound data access methods are provided in listing 3.

Negating an interval can be done without introducing any further error since it simply requires flipping the sign bit. Its implementation therefore simply swaps the upper and lower bound after negating them, as shown in listing 4.

```
interval::interval()
: lb_(0.0), ub_(0.0)
{
}

interval::interval(const double x)
: lb_(x), ub_(x)
{
  add_error(lb_, ub_);
}

interval::interval(const double lb,
                   const double ub)
: lb_(lb), ub_(ub)
{
  static const double i
     = std::numeric_limits<double>::infinity();

  if(lb==lb && ub==ub && (lb!=ub ||
     (lb!=i && lb!=-i)))
  {
    if(lb_>ub_)  std::swap(lb_, ub_);
  }
  else
  {
    lb_
      = std::numeric_limits<double>::quiet_NaN();
    ub_
      = std::numeric_limits<double>::quiet_NaN();
  }
}

double
interval::lower_bound() const
{
  return lb_;
}

double
interval::upper_bound() const
{
  return ub_;
}
```

Listing 3

```
{
  lb_ = -lb_;
  ub_ = -ub_;
  std::swap(lb_, ub_);
  return *this;
}
```

Listing 4

Comparing **interval**s is a little more difficult. It’s clear enough what we should do if the two **interval**s don’t overlap, but it’s a little confusing as to what we should do if they do.

We could take the position that two **interval**s shall compare as equal if it is at all possible that they are equal. Unfortunately, this means that equality would no longer be transitive. An **interval** *x* might overlap *y* and *y* might overlap *z* despite *x* not overlapping *z*. We would therefore have to be extremely careful when dealing with equality comparisons. A simpler alternative that maintains transitivity of equality is to compare the midpoints of the intervals.

We shall use the second approach and its implementation is given in listing 5. Note that we must also provide an **is_nan** method to allow the comparison operators to behave correctly in their presence.

When adding **interval**s we add their upper and lower bounds and then add further errors to the bounds, as shown in listing 6.

Subtracting **interval**s is similarly straightforward, as shown in listing 7.

When multiplying **interval**s we must consider the effects of mixed positive and negative bounds. Specifically, we have 9 cases to consider

$(-x_l, -x_h) \times (-y_l, -y_h)$

$(-x_l, -x_h) \times (-y_l, +y_h)$

$(-x_l, -x_h) \times (+y_l, +y_h)$

```
bool
interval::is_nan() const
{
  return !(lb_==lb_);
}

int
interval::compare(const interval &x) const
{
  const double lhs =   lb_*0.5 +   ub_*0.5;
  const double rhs = x.lb_*0.5 + x.ub_*0.5;

  if(lhs<rhs) return -1;
  if(lhs>rhs) return  1;
  return 0;
}
```

Listing 5

Unfortunately interval arithmetic is not entirely foolproof

```
interval &
interval::operator+=(const interval &x)
{
  lb_ += x.lb_;
  ub_ += x.ub_;

  add_error(lb_, ub_);

  return *this;
}
```

Listing 6

$(-x_l, +x_h) \times (-y_l, -y_h)$
$(-x_l, +x_h) \times (-y_l, +y_h)$
$(-x_l, +x_h) \times (+y_l, +y_h)$
$(+x_l, +x_h) \times (-y_l, -y_h)$
$(+x_l, +x_h) \times (-y_l, +y_h)$
$(+x_l, +x_h) \times (+y_l, +y_h)$

where each x and y value is greater than or equal to zero.

Rather than consider each case individually, I propose that we simply calculate all 4 products and pick out the least and the greatest. We can find the least of them reasonably efficiently with the first pass of a crude sorting algorithm. To find the greatest we can assign, rather than swap, values since we shan't subsequently need the in-between values.

Note that if either argument is NaN we simply set the result to NaN. Having tested for this case, the only situation in which we shall come across NaNs is if we multiply an infinite bound by a zero bound, in which case we replace it with zero to reflect the fact that infinite bounds are open.

Listing 8 provides an implementation of multiplication.

When dividing, we must ensure that we properly handle denominators including zeros and infinities. Note that, similarly to how we did for multiplication, we exploit the fact that a NaN result from non NaN arguments identifies that both were infinities. We further exploit the fact that the lower bounds cannot be plus infinity nor the upper bounds minus infinity to replace such NaNs with the correct infinite bounds.

The cases of division by zero are handled by explicit branches and, as we did for multiplication, we find the lower and upper finite bounds with our crude sorting algorithm as shown in listing 9.

```
interval &
interval::operator-=(const interval &x)
{
  lb_ -= x.ub_;
  ub_ -= x.lb_;

  add_error(lb_, ub_);

  return *this;
}
```

Listing 7

Aliasing

Unfortunately interval arithmetic is not entirely foolproof. One problem is that it can yield overly pessimistic results if an interval appears more than once in an expression. For example, consider the result of multiplying the interval (-1, 1) by itself.

```
interval &
interval::operator*=(const interval &x)
{
  if(!is_nan() && !x.is_nan())
  {
    double ll = lb_*x.lb_;
    double lu = lb_*x.ub_;
    double ul = ub_*x.lb_;
    double uu = ub_*x.ub_;

    if(!(ll==ll))  ll = 0.0;
    if(!(lu==lu))  lu = 0.0;
    if(!(ul==ul))  ul = 0.0;
    if(!(uu==uu))  uu = 0.0;

    if(lu<ll)  std::swap(lu, ll);
    if(ul<ll)  std::swap(ul, ll);
    if(uu<ll)  std::swap(uu, ll);

    if(lu>uu)  uu = lu;
    if(ul>uu)  uu = ul;

    lb_ = ll;
    ub_ = uu;

    add_error(lb_, ub_);
  }
  else
  {
    lb_
      = std::numeric_limits<double>::quiet_NaN();
    ub_
      = std::numeric_limits<double>::quiet_NaN();
  }

  return *this;
}
```

Listing 8

This can be **particularly troublesome** if such expressions appear as the denominator in a division

Ignoring rounding error, doing so yields

$$\begin{aligned}(-1,1)\times(-1,1) &= (\min(-1\times -1, -1\times 1, 1\times -1, 1\times 1),\\ &\quad\ \max(-1\times -1, -1\times 1, 1\times -1, 1\times 1))\\ &= (-1,1)\end{aligned}$$

rather than the correct result of (0,1).

This can be particularly troublesome if such expressions appear as the denominator in a division. For example, given

$$\begin{aligned}x &= (-1,1)\\ y &= \left(\frac{1}{2},1\right)\\ z &= (0,1)\end{aligned}$$

and again ignoring rounding errors, we have

$$\begin{aligned}\frac{z}{x\times x+y} &= \frac{(0,1)}{(-1,1)+\left(\frac{1}{2},1\right)}\\ &= \frac{(0,1)}{\left(-\frac{1}{2},2\right)}\\ &= (-\infty,\infty)\end{aligned}$$

rather than

$$\begin{aligned}\frac{z}{x\times x+y} &= \frac{(0,1)}{(0,1)+\left(\frac{1}{2},1\right)}\\ &= \frac{(0,1)}{\left(\frac{1}{2},2\right)}\\ &= (0,2)\end{aligned}$$

```
interval &
interval::operator/=(const interval &x)
{
  static const double i
     = std::numeric_limits<double>::infinity();
  if(x.lb_>0.0 || x.ub_<0.0)
  {
    double ll = lb_/x.lb_;
    double lu = lb_/x.ub_;
    double ul = ub_/x.lb_;
    double uu = ub_/x.ub_;

    if(!(ll==ll))  ll = i;
    if(!(lu==lu))  lu = -i;
    if(!(ul==ul))  ul = -i;
    if(!(uu==uu))  uu = i;

    if(lu<ll)  std::swap(lu, ll);
    if(ul<ll)  std::swap(ul, ll);
    if(uu<ll)  std::swap(uu, ll);

    if(lu>uu)  uu = lu;
    if(ul>uu)  uu = ul;

    lb_ = ll;
    ub_ = uu;

    add_error(lb_, ub_);
  }
```

Listing 9

```
  else if((x.lb_ <  0.0 && x.ub_ >  0.0)
       || (x.lb_ == 0.0 && x.ub_ == 0.0)
       || (  lb_ <  0.0 &&   ub_ >  0.0))
  {
    lb_ = -i;
    ub_ = i;
  }
  else if((x.lb_ == 0.0 && lb_ >= 0.0)
       || (x.ub_ == 0.0 && ub_ <= 0.0))
  {
    lb_ =  0.0;
    ub_ = i;
  }
  else if((x.lb_ == 0.0 && ub_ <= 0.0)
       || (x.ub_ == 0.0 && lb_ >= 0.0))
  {
    lb_ = -i;
    ub_ =  0.0;
  }
  else
  {
    lb_
      = std::numeric_limits<double>::quiet_NaN();
    ub_
      = std::numeric_limits<double>::quiet_NaN();
  }
  return *this;
}
```

Listing 9 (cont'd)

there are many ways in which we might represent numbers with computers and they all convey certain strengths and weaknesses

If we wish to ensure that such expressions yield as accurate results as possible we shall have to rearrange them so that we avoid aliasing. For example, rather than

```
x*x + 3.0*x - 1.0
```

we should prefer

```
pow(x+1.5, 2.0) - 3.25
```

Provided we keep in mind the fact that interval arithmetic can be pessimistic we can still use it naïvely to give us a warning of possible precision loss during a calculation.

Unfortunately there is a much bigger problem that we must consider.

Precisely wrong

Consider the use of our `interval` type in the calculation of the derivative of the exponential function at 1. The listing snippet below illustrates how we might calculate it for some leading number of zeros in δ, `i`.

```
const interval d(pow(2.0, -double(i)));
const interval x(1.0);
const interval df_dx = (exp(x+d) - exp(x)) / d;
```

Note that since both *x* and δ are powers of 2, this code is needlessly pessimistic; they and their sum have exact floating point representations.

A more accurate approach is:

```
const double d = pow(2.0, -double(i));

const interval  x(1.0,   1.0);
const interval xd(1.0+d, 1.0+d);

const interval df = exp(xd) - exp(x);
const interval df_dx(df.lower_bound()/d,
                     df.upper_bound()/d);
```

Note that in the final line we are also exploiting the fact that a division by a negative power of 2 is also exact when using IEEE floating point arithmetic.

Figure 1 reproduces the graph of the error in the numerical approximation of the derivative together with the precision of that approximation. Specifically, it plots minus the base 2 logarithm of the absolute difference in the approximate and exact differentials, roughly equal to the number of correct bits, against minus the base 2 logarithm of δ, roughly equal to the number of leading zeros in its binary representation. On top of this it plots minus the base 2 logarithm of the difference between the upper and lower bounds of the approximate derivative, which is a reasonable proxy for the number of bits of precision in the result.

Clearly, there's a linear relationship between the number of leading zeros and the lack of precision in the numerical derivate. Unfortunately, it is initially in the opposite sense to that between the number of leading zeros and the accuracy of the approximation.

We must therefore be extremely careful to distinguish between these two types of error. If we consider precision alone we are liable to very precisely calculate the *wrong* number.

So, whilst intervals are an extremely useful tool for ensuring that errors in precision do not grow to significantly impact the accuracy of a calculation, they cannot be used blindly. As has consistently been the case, we shall have to think carefully about our calculations if we wish to have confidence in their results.

I shall deem this type a roast duck; tasty, but quite incapable of flight.

Mmmm, quaaack.

You're going to have to think!

We have seen that there are many ways in which we might represent numbers with computers and that they all convey certain strengths and weaknesses.

We have found that none of them can, in general, remove the need to think carefully about how we perform our calculations. For certain mathematical calculations, the algorithms we use to compute them and the implementation of those algorithms are by far the most important factors in keeping errors under control.

In consequence, I contend that double precision floating point arithmetic, being an efficient and parsimonious means to represent a vast range of numbers, is almost always the correct choice for general purpose arithmetic and that we must simply learn to live with the fact that we're going to have to think! ■

Further reading

[Boost] http://www.boost.org/doc/libs/1_43_0/libs/numeric/interval/doc/interval.htm

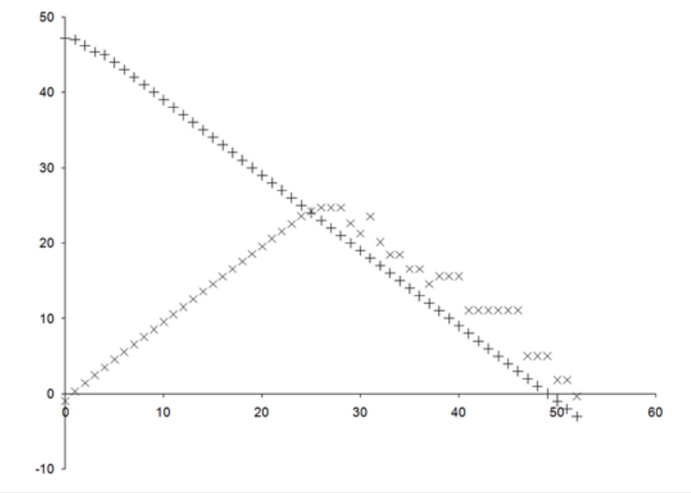


Figure 1

Systems Thinking Software Development

Many processes cause more problems than they solve. Tom Sedge shows how to tailor your own.

For a long time I've been searching for better ways to go about the business of software development, particularly when what I thought were successful developments, produced on time and budget, still encountered many problems in production. Sometimes it has seemed as if it is only possible to know what should have been built after several years and versions getting it wrong. Before I came across Systems Thinking it hadn't occurred to me that perhaps I was looking in the wrong place, forever focussing on the design and coding process, requirements or methodology and not the bigger picture of the problem I was trying to solve.

This article isn't just aimed at managers and decision-makers in software companies. It's aimed at senior developers, architects and team leads too. Towards the end, I'll explain how you can make use of Systems Thinking even if you work in an environment that may be largely beyond your control, and give you some pointers on how you might be able to persuade the decision-makers to give it a chance.

Please note that throughout this article I am presenting my own personal understanding and explanations of the subject. Please do explore the links and references in the endnotes for other perspectives.

The state of software development

There's much that is good about software development today, and yet despite the issue being highlighted time and time again over many years, the industry does continue to suffer from failure, cost and time overruns and poor quality. From high level government projects, like the NHS programme for IT and systems at HMRC, down to medium and even small scale private sector work, there are problems. The chances of a project being completed on time, on budget, fit for purpose and with high quality are far smaller than they should be when we consider the sheer quantity of new languages, tools, and working practices.

Over the years attention has turned to a number of candidates for blame:

- Requirements and designs are too complex or too simple. They are over-specified or under-specified, or may be unclear and inaccurate, and they come in late, and shift and change too much and too often.
- Code is poorly written and doesn't follow standards. It isn't modular enough or may be too modular. Perhaps there's not enough object-orientation, perhaps too much, and the code can be buggy, hard to read, under- or over-commented and always insufficiently tested.
- Documentation may be absent, out of date and often incomplete. Sometimes it is excessive, excessively unclear or otherwise insufficient and inaccurate.
- A software development process can slow progress, generate too many useless documents and too few useful ones. The process in use is often good at handling either stable or changing conditions, but not both. It may involve too little or too much planning and impose excessive or insufficient controls.

To address these there have been many good and useful things that promise to help, some technical and some managerial:

- New languages that promise to eliminate bugs and raise quality.
- A zillion third-party libraries that promise to take the effort out of coding, and allow a week's work to be done in a day.
- New all-singing and all-dancing development tools which can analyse code, eliminate bugs and support all types of refactoring.
- New capabilities in collaborative source code control systems supporting multiple and overlapping lines of development.
- We've also seen practices that promise to raise quality and productivity. Unit testing, pair programming, collective code ownership, code reviewing, design modelling, use cases, user stories and more.
- There have been new processes that promise to raise quality, dramatically speed development and increase agility. A tidal wave of agile methodologies from RUP (if you class it as agile, many don't), through Lean, XP, Scrum, and more recently Kanban and half-a-dozen others.

Given the demonstrable value of many of these individual improvements, why aren't there more successes? There are several reasons why I think we haven't yet reached the promised land where the vast majority of software developments are an unqualified success.

Understanding and improving work

The purpose of any software development is to provide software to a group of end users to make their lives easier. In some cases those developing the software have no real understanding of the work that those end users do. A clear understanding of their work, how it flows and the needs that drive that work is essential for good requirements and subsequent good design. All too often there's a vague set of requirements to computerise an existing manual workflow which risks exactly that: computerisation with no other improvement to how the work is done. In other cases the task may be to replace an inadequate computer system with a slightly better one, but one which still implements the same flawed approach.

I remember from my days using Lotus Domino that a designer somewhere had clearly thought that a good way to computerise a filing cabinet was to replicate the same inflexible concepts of cabinets, binders, folders and files in computer form. Perhaps they thought it would at least be easy to understand? What they achieved was all the disadvantages of the paper system plus some minimal benefits. Worse than that, the restrictive paper concepts made little sense in computer land, so they also created confusion. No-one seems to have asked the question: what are users actually trying to do?

Tom Sedge is an independent coach on management and communication, following time as a developer, architect and prgramme manager. His current focus is helping people start their own businesses. He can be contacted at: http://www.timelesschange.co.uk

Consulting customer proxies and power users can hinder as well as help because they are often not representative of the main user base

It doesn't matter how well written the requirements, how many focus groups or workshops, how many or few features are included or how polished the UI. None of these will make a difference if there's a failure to understand the system of which the user is part. Consulting customer proxies and power users can hinder as well as help because they are often not representative of the main user base and as experts may have quite a different approach to the work than their peers.

The best way to get this understanding is to go there and see for oneself. It is too important to trust to a third-party who might jump straight into solution mode and deliver a specification for the wrong thing. Key staff need to go and meet users in their workplace and understand them and what they do. It is important to include developers in this, not just requirements and user-interface experts, because developers need the same understanding if they are going to model concepts accurately and produce a clean and appropriate design for a software solution.

Once there's understanding of how the work is performed currently, careful thought needs to be given to whether the introduction of new software could and should change that work. Considering changing how users work should be an integral part of any software development process. There may well be some customers that don't want that, but the greatest benefits of any technology lie in complementing the technology with an optimised workflow, so it is worth trying to persuade them. Without an effective and appropriate workflow even the best and most technically beautiful software may be dead on arrival.

I am suggesting that one of the principal barriers to greater success is a failure to properly understand the nature of the work that end users do, the system of which that work is part, and how that system can be changed and optimised through the introduction of new software. Think about all those useful little applications that were written by people wanting to solve their own problems. When they fully understand the nature of their problem, then they can solve it in elegant ways that dramatically improve productivity.

Misunderstandings start before we get to any detailed requirements, in the basic knowledge of the system we're trying to improve. The first customer questions shouldn't be 'What do you want and when?', they should be 'How do you work now and what are you trying to achieve?'. We need a mechanism for understanding and mapping how people work, uncovering their needs, and a design language for workflows that allows us to design changes and predict their impact.

Trialling and measuring changes

When a change to a system is being made, there are always risks. Will the change be a change for the better? Did we really do a good job with that software and what's the evidence for that apart from a thank you from the customer?

During the process of designing software and optimising workflow, there'll be many ideas and possibilities. The best way to choose between alternatives is to try them out with real users in their normal work, and not just to rely on their opinions but to quantitatively measure the impact. That way good can be distinguished from bad, the whole solution can be improved, and the effect on the customer of the final solution can be estimated before most of the work is done, giving real support to the business case.

The popularity of prototyping varies, but it is rare to trial prototypes with real users and even rarer to test them with real work in a live situation, even though there are a number of ways of doing this safely, for example using the prototype in parallel with the existing solution. There's rarely any attempt to measure the impact in quantitative terms.

The second suggestion I make is that the lack of proper trials and measurements of solution effectiveness means that even with a good understanding of how users work, poor solutions can still be delivered. We need a mechanism for defining sensible measures and trying out changes in order to make the right decisions about how software should work.

The process of developing software

When designing and building software, there needs to be a way of working that maximises the value delivered to the customer while minimising costs. It turns out that to do this well there needs to be the same kind of understanding about internal processes as for customers and how they work. This goes far beyond just considering developers and testers, it needs to include everyone who is involved in the production and maintenance of software including sales, marketing, product management, support and quality management.

How does the end-to-end software development process work in your company? How can it be improved and how can these improvements be trialled and measured? Without some way to measure improvements, ideas are sometimes tried blind in the hope that some will be winners. There may be a thousand suggestions and recommendations out there, but which ones apply in any specific situation?

A classic mistake is to presume that a software development process should be fixed and that it comes in neat off-the-shelf bundles labelled as 'methodology X'. Without exception, every workplace has some unique needs, and though a methodology might indeed provide useful tools and ideas, maximum effectiveness demands customising the way work is done to meet the environment. This is only common sense. A software development process can slow down work in several ways. It can be so prescriptive and burdened by ceremony that it stifles through workload, or it can be so free and lightweight that it impedes progress through lack of direction, focus and ensuing re-work and waste. Agile is no silver bullet here, and while it may addresses some concerns of software developers it may not help with those of other departments [Kelly11]. Indeed many of the agile methodologies have little to say about people who are not developers or testers.

My third suggestion is that a lack of understanding about the effectiveness of our own software development processes including how to measure them and safely improve them, is a major factor that gets in the way of delivering good software to customers.

think of it as a way to **craft your own personalised silver bullet** of a process that ideally suits how **you and your team need to work**

Enter systems thinking

So what is needed is a solution that provides a template for working with and understanding customers and their work, analysing and improving workflows, trialling and measuring the impact of improvements, and which can also be applied into the internal processes used to create software. A solution which is lightweight, based on facts, fast, implements continuous improvement, and is low-cost and low-risk. As you might have already guessed, the process I'm going to suggest that can help with all of this is Systems Thinking.

Systems Thinking is a term that was coined in the 1950s and later grew out of the work of W. Edwards Deming whose book *Out of the Crisis* [Deming82] condemned the state of modern management. His ideas were taken up by Taiichi Ohno in Japan, who went on to design the Toyota Production System; a systems thinking approach to the manufacture of cars which simultaneously reduced costs and raised quality. It was the Toyota Production System that directly lead to Toyota's growth and dominance in car manufacturing. More recently John Seddon [Seddon], author of *Freedom from Command and Control* [Seddon03], has pioneered applications of the same approach to service industries and further developed it.

Systems Thinking is an approach to work that considers the whole system of which the work is a part and provides tools to map out and improve that system. This is in contrast to traditional approaches that tend to focus on micro-optimisations in particular areas. It can be applied to any organisation producing products or services, and includes tools to study and learn from demand, design measures of performance, and map out processes. It then guides the design of changes to those processes and safely trials changes using measures to prove their effectiveness before they are rolled out.

It works through a continual cycle of improvement, that can be driven at whatever speed an organisation needs, delivering low-risk incremental changes that add up to radical long-term streamlining and optimisation. It does this with staff experiencing both minimal disruption and maximal involvement in designing improvements, which has the side effect of significantly raising morale and motivation because those who do the work get to shape how the work changes.

The problem with micro-optimisations are that improvements in one area may cause bigger problems elsewhere. The improvements they appear to deliver (for example cost saving) may actually harm customers of the product or service somewhere else, leading to poorer quality, worse service and ultimately damage to reputation and lower sales. This wrong-headed strategy is something we each encounter every day. For example, I regularly have the 'joy' of interacting with BT's computerised helpline, which not only forces me to laboriously beep my way through a deep hierarchical menu, but now wants me to speak my request so I can enjoy a seemingly endless tennis match of: 'Did you say X?', 'No I said Y!'. Companies like BT choose to annoy their customers in the name of small cost savings, causing themselves greater costs elsewhere handling complaints, lost business and effects on reputation. If they saw the true costs, I'm sure they wouldn't do it.

I don't present Systems Thinking as a silver bullet. Instead, think of it as a way to craft your own personalised silver bullet of a process that ideally suits how you and your team need to work. You'll create a flexible silver bullet that you'll continue to polish as your circumstances change. That's the only guarantee of success: having the knowledge, confidence and tools to tailor your approach to changing conditions.

Let's take a look at the main components of Systems Thinking, which I present through the lens of the Learn. Think. Do. improvement cycle, which I also use as a model for my coaching work. (You'll find that Seddon calls this process Check, Plan, Do. I prefer my own terminology, but in essence it is the same idea.)

You'll notice that Systems Thinking provides a continuous cycle of improvement. There's a learning phase, where we study and understand customer's needs, their systems and environments (demand), define new ways to measure performance, and map out processes. This is followed by a thinking phase, where we design improvements to these processes, and a doing phase where these improvements are trailed and, if successful, made permanent. Let's look at it in more depth, and specifically how each stage can be applied to Software Development.

Learn

The first and biggest stage is Learn. This is split into three parts: learning about demand, designing improved measures, and mapping out processes.

Throughout all of these stages, the way I recommend you work is to establish a multi-disciplinary improvement team, with representatives from every department involved in delivering your product or service, and that you include experienced front-line staff with expert knowledge of what actually happens on the ground.

Demand

In Systems Thinking, demand is the name you give to all contact from customers, whether they are for delivered products and services, support calls, complaints, enquiries, future developments, sales, or any other purpose.

There are two types of demand:

- Value demand – anything your team does that gives the customer value. This is what you're in business for.
- Failure demand – anything you do that doesn't give the customer value. This includes any waste or failings on your team's part that get in the way of providing value demand.

Companies that have a high level of failure demand have high costs servicing that demand. It is quite common for failure demand to be responsible for a significant percentage of all costs. If you want to dramatically reduce costs, do it not by focussing on costs like BT has done, but by focussing on reducing failure demand.

In Software Development, examples of value demand include:

- The main software products and services you create that solve a customer's problem and makes users lives easier.

in a competitive landscape **burdening your customers with these extra costs** will only make the competition more attractive

- Any supporting products and services that you offer which contribute to solving their problems, excluding any problems you created – for example customer difficulties installing or configuring your products that leads you to sell consultancy services on product installation is an example of failure, not value.

Examples of failure demand include:

- Most customer support. This may be due to inadequate product quality, lack of fitness-for-purpose, too much complexity, lack of ease-of-use, poor documentation, or other failings.
- Most customer training. This may be due to product complexity, lack of ease-of-use or fitness-for-purpose.
- Most customisation and installation services, again due to complexity, lack of documentation or ease-of-use.

You might be surprised to see some of these on the list as ‘failure demand’. It is very easy to become so accustomed to handling failure demand that these activities become taken for granted. I’m not suggesting that you simply dispense with support and training or that they are unnecessary. What I’m saying is that perhaps you wouldn’t need to provide nearly so much of both if you considered the causes of this demand and improved your software and working practices to reduce them.

It can be tempting to look at these services as good sources of additional revenue, but in a competitive landscape burdening your customers with these extra costs will only make the competition more attractive. Look at the success of Apple and its App Store: redesigning applications to make them simple, intuitive, instantly deployable and updatable, and with minimal need for documentation or support has enabled thousands of independent developers to go solo. They are no longer burdened with the failure demand costs that in the past would have made this route impractical.

From your point of view, there are two places where value and failure demand operate:

- External demand: This is demand arising from customers external to your organisation, of which I gave some examples above.
- Internal demand: This is demand arising inside your own organisation in the service of external demand. It is the things you do, including your own internal processes, to provide your products and services.

It is desirable to maximise internal value demand and minimise internal failure demand, because that means maximum focus on serving the customer at minimum cost. Examples of internal demand will vary from business to business, but there are common indicators that I would suggest are associated with a good balance:

- Communication is clear, concise, and to the point. There’s culture of open and direct communication person-to-person rather than up and down the hierarchy.
- Meetings focus on making decisions rather than exploring options and they are short and infrequent.
- There are frequent whiteboard sessions with a range of staff from different departments, so knowledge about the whole business is spread widely and in considerable depth. Priorities are clear, long-term and rarely change.
- Everyone can contribute ideas and suggestions for improvement, no matter how junior, and there's a culture of question-asking and fact-finding before decision-making.
- Role-rotation, job-shadowing and other techniques are used to share skills and experience and reduce over-reliance on a few key individuals.
- There’s rapid learning from failure through a tight cycle of feedback and continuous improvement, and change is driven through the focus of providing customer value.
- There’s an appropriate level of formal process and internal documentation. Appropriate means it is all used, all useful, and all serves a direct customer purpose.
- Members of the core development team are in close contact with customers and regularly visit their workplace.
- Internal IT is streamlined, simple and appropriate to needs.

This is my own list and is far from an exhaustive or definitive, but it should give you some idea of how close your organisation is to serving demand well. If you have many of these, be happy; you’re probably doing quite well. If you have few or none, also be happy; there’s big scope for improvement. If you have few of these and don’t think others will let you make the changes needed to get more, then either find a way to convince them or move on. I’m a strong believer that life’s too short to let other people waste it.

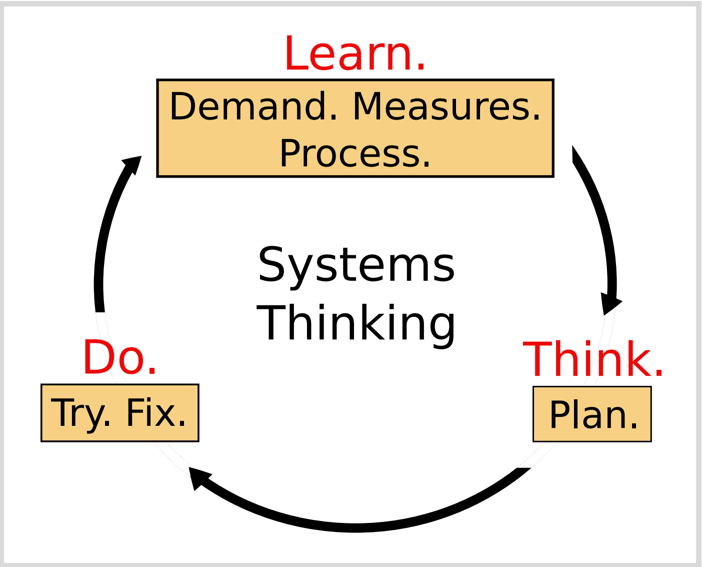


Figure 1

help your customers understand their own demand and improve their business too, providing extra value to them

Studying demand involves a lot more than just listing types. You'll need to examine each type in detail to understand how much you have of it, where it comes from, what the underlying causes are, and how the way you work is contributing to it. Once demand is well understood, the goal is simply to maximise value demand and minimise failure demand. Some demand is predictable and some is not. Systems Thinking helps with both because enables us to optimise processes to handle predictable demand, whilst increasing flexibility to handle the unpredictable.

Studying demand can be done from both the software provider's perspective (i.e. studying what your customers want from you), and if you are a B2B company, from your customer's perspective (i.e. studying what your customer's customers want from them and how you can help them provide it). So learning about demand can be used by you not only to improve your own business, but also help your customers understand their own demand and improve their business too, providing extra value to them that will help distance you from the competition.

Measures

The next step is to design some new measures (or if you prefer: metrics). We talk about measures deliberately, since they always follow events – you are measuring something that has already happened. In order to know the quantity of value demand and failure demand in your software company, you need a reliable way to measure it. Without that, how will you be sure that you've made any improvement?

The alternative, 'targets', are not a good idea since there's no reliable way to decide what a target should be, and the presence of a target can severely limit both the ambition and scope for improvement. Ambition is limited because targets are usually set quite low, with a modest change in mind. Scope is limited because targets are often set for things that aren't closely related to delivering value to customers, and then the goal becomes distorted into meeting the target rather than providing the best product or service.

Systems Thinking provides a way to design effective measures so that you can track current and future performance. In software some of this information can be drawn from existing failure and value demand systems – for example bug trackers, feature requests, customer support systems and sales databases. Once established measures are tracked over time to see how they are affected by changes.

All of your measures need to be relevant and important to your customers. Generic examples might include the end-to-end time it takes you to deliver new features or bug fixes, how quickly you respond to enquires, or the number of severe bugs customers detect for you. Specific examples will include measures relevant to a particular product and customer: the impact your software has on them.

None of those I've mentioned are 'off-the-shelf' measures that you should necessarily adopt. The whole point is to work out which measures would best suit you and your customers. The important point is that they must all be things that your customers care about. With a clean set of measures that directly represent how well you are providing value to your customers, you have a benchmark against which you can test any change you make to your workflow.

If your customers have their own customers, you can also help them to come up with measures that will demonstrate the value your software gives to their own customers. There's nothing like concrete evidence of the difference you made to their bottom line to convince them to stay with you for the long term.

Process

The final step in Learn is to map out your internal workflow. I've been experimenting with a simple visual process language that draws out the important stages and hand-offs involved in a workflow. This picture allows everyone in your organisation to understand the whole system, explains much of the origin of your failure demand, and is also used later on in Do to design and prototype improvements.

Mapping processes is best done in collaborative face-to-face workshops using whiteboards, flip-charts, index cards, post-it notes and other low-tech tools. The workshop should include a representative of everyone from every department involved in the design and delivery of software, bringing together expertise to collectively understand how work is done. There have been reports of many surprising things learned through this approach, particularly in organisations that have separate departments with poor communication. It is also all too common for duplicate and unnecessary steps to become apparent.

This same workshop approach can then be used when working with customers to map out and understand their processes, so that you get a full understanding of their workflow before you start designing software. This avoids prematurely discussing requirements before the customer environment and specific needs are understood.

Think

The second stage is to design workflow improvements using the process maps created in Learn.

This is where planning takes place, but it is not what you might be used to as planning. Instead of big Word documents, Gantt charts, MS Project plans or similar tools, the same form of collaborative workshop is used to design changes to the process. Changes are simulated by creating new steps, removing steps, altering the flow of work, handoffs and work products that are created. Each change is then considered end-to-end and the workshop team look for possible problems or unintended consequences.

This is a form of low-cost process prototyping that flushes out the best ideas, works out their details and then stress-tests them on paper before you go near trying them out. The result of Think is one or more concrete improvements that have been analysed from the perspective of the whole system and how they will deliver customer value. Within your own software development workflow, it is likely that different development methodologies will be discussed. That's fine, so long as you treat each one as a template rather than a prescription and are prepared to properly methodology is much less important than people usually suppose

simulate the full consequences using your process map before adopting anything.

I don’t want to either promote or bash any specific methodologies, since they all have strengths and weaknesses, and often contain very useful tools. Methodologies generally split into high-discipline and low-discipline groups. High-discipline methodologies come with an array of constraints that will impact your business elsewhere. Whether the impact is positive or negative you’ll have to decide. Examples include Waterfall, XP and Scrum.

Although I am far from a fan, Waterfall can work in some situations, with simple and clear requirements and short projects. XP contains a tightly-prescriptive set of practices which also dictate how you approach requirements, how you deal with your customers, and changes you’ll need to make in your working environment and support systems. If you’re willing to change your business around XP and XP fits (once you’ve studied and understood your process and the impact it will have), then go for it. Just make sure all the stakeholders are on board with that decision, not just the development team. Scrum is less prescriptive than XP, but will force you to organise your business around set-length sprints. Again, so long as you are happy with the consequences, and everyone in the business can buy into the sprint cycle, then go for it.

Examples of low-discipline methodologies include Lean and Kanban. These are mostly concerned with promoting continual flow and improvement and in general they are more flexible than the high-discipline alternatives. However, precisely because they are so simple, you’ll need to design a layer on top of your own processes to create a full recipe for how your team develops software. So low-discipline does require more work on your part, albeit for a more bespoke result.

I find methodology is much less important than people usually suppose, and the best methodology in the world will still lead you to failure if you are only thinking about the mechanics of writing software and not the wider context of its usage.

The goal of Plan is not to re-write your process or procedure documents. That’s something that you should only do when you are ready to make a change permanent in Do. Plan is all about designing the best process you can to deliver software.

You might require a special permit to do so, but you probably don’t want to be discussing issues of process compliance or standardisation at this stage, and it may be best to keep those guys out of the room. That’s because Plan is about exploring options and compliance people tend to be better at critiquing detail. Obviously if do you have a good idea what they need, then you can consider it now. They do need to become involved at some point as usually there’s an important business reason behind the compliance function. But there’ll be time for that later when you have both specific changes and a strong line of reasoning about how those changes are the right thing to do.

With a customer’s workflow, you adopt the same approach. Having mapped out their workflow with them, work alongside them to see how it can be improved. Do this before you start thinking about the software. Just focus on what the users are trying to do and how that can best be done. When you are both comfortable with the new workflow, then starting thinking about what the software will need to do. Resist jumping into features and benefits too early. The goal isn’t to find everything your software could possibly do. It is to streamline the customer’s workflow and only then create the minimal amount of software that is needed to make that flow run smoothly. You’ll end up writing less software that is a better fit. Your costs will go down and your customers will get a cheaper and better result more quickly.

Do

The final stage in the improvement cycle is Do. Here you get to try out changes and make them permanent.

The traditional idea of special ‘change’ projects is an unhelpful one, because it pretends that change should be something infrequent and separate from normal work, rather than constant and an integral part of a team’s DNA. Treating change as a project can actively inhibit progress. It needs to be small, incremental and continual. Why wait until the end of a project to learn?

Try

The Try step is all about testing the water with minimal costs. Rather than simply take your best ideas from Plan and hope they work, Try allows you to gather evidence and refine or reject them, reducing your risks and saving on the costs of change. In Try, you use your improvement team to identify the best way to try out a change, with minimal costs. This can involve a temporary paper or card-based solution to allow the new method to be tested and refined.

Software people usually hate this, because their whole job is to create computerised solutions, but the important point to remember is that you need the ability to adapt and refine your solution in real-time, and you are very unlikely to have the time and the money needed to develop software prototypes that will quickly be superseded. Try needs to be cheap, simple and fast.

Changes in your own workflow are trialled for however long it takes for the improvement to show up in your measures, perhaps as a reduction in internal bug counts or faster turnaround times for new features. If no improvement shows up, then you need to have the courage to reject the change and go back to the drawing board. Either it hasn’t worked or your measures were not good enough to detect the improvement. If you suspect the latter, go back and improve your measures before going any further.

If you want to create the best software for your customers, you need to trial it with them too. This means much more than just throwing a Beta over the wall. You need to sit with customer users and watch them use it and learn how how to improve it as a result. Best of all is to wait for trailed improvements to show up in their measures so that you have concrete evidence you're on the right track.

Note that sometimes for whatever reason, it isn’t possible to do Try. Perhaps the cost would be too great or there isn’t enough time. That’s ok, you can move straight from plan to Fix, just be aware that you are putting a lot of pressure on your planning being correct and it might not be. Seddon doesn't talk about a separate Try step, but I think it makes sense as a way to limit risk, so long as the circumstances permit it.

Fix

This is all about making changes permanent, having become convinced of their effectiveness in Try (or Plan if you are skipping Try).

When it comes to your process and procedure documentation, this is the time to draw new versions because you have refined your workflow changes and they are ready to go live. It is also the time to fully engage your compliance people (if you are lucky enough to have them!) and start the discussion on how to demonstrate compliance with ISO9001, CMM, TickIT or anything else that you subscribe to. Make sure that you only look to provide evidence to demonstrate compliance and do not end up changing your workflow purely in response to compliance demands. If you do decide to tweak your changes in any significant way, go straight back to Plan, properly consider the change, trial the new version again in Try and only then go forward.

If your compliance people insist on a specific change, remind them that their industry is always claiming that they neither prescribe or proscribe how you work, and ask them to refer you directly to the rule in the standard that they are concerned about. Read the rule, work out how to provide evidence for it and then show them both the rule and your justification of how your change meets it.

When it comes to supporting IT systems that might need to be replaced or changed, software companies have a big advantage over others as they have the option to roll their own. I always recommend seriously considering this, because you can create a streamlined software solution that precisely meets what you need, thus minimising sources of internal failure demand. Though it might be tempting to use an off-the-shelf product, they are often so feature-full and generic that they do far too much, are difficult to configure, and the high price that they justify through the feature set is out of proportion to the small number of things you actually want to use. Of course, there are times when it makes sense to buy off-the-shelf; source-code control systems are a good example (and the best of these are free in my experience), because it would be impractical to write your own.

Having completed Do, you then embark on a new cycle of Learn, to further refine your understanding of demand, measures and process, and to enable you to respond to changes in your marketplace. If you really want to deliver the best software you can to your customers, you'll build a long-term relationship with them and keep going through cycles of improvement together.

The cycle repeats over and over again, leading you to dramatic long-term improvements with low-risk, reducing costs and raised morale. That's the promise of Systems Thinking.

Getting started

I promised in the introduction that I'd say more on how to get started with Systems Thinking, particularly if you work somewhere where how the work is done is largely not under your control.

You could apply it to a single product or project, properly including customer end users as a limited experiment in trialling new working practices. Or you could apply it within a single department or team, so instead of considering the whole of your organisation, your customers become the other departments or groups that need things from you. You would go through the same process of studying your demand, defining useful measures of your work's quality and productivity in terms of other team's needs, mapping your processes and trialling and improving them. While this won't put you closer in touch with end users, you could use it to build a documented track-record of internal success and then use that to raise awareness and gain buy-in to take it further.

You could apply it within a team, perhaps just with developers. Now perhaps your customers are the project manager and QA and the things you measure will be the things they care about. If you really have little autonomy, you could apply to yourself personally and look at how you might both measure and improve how you do your work. If you build a body of evidence from this, that could be quite persuasive at your next appraisal both for your own advancement and for Systems Thinking.

Measures are a great way to get support for change. If you can devise better measures and go and gather the data, you can use that as evidence to persuade other people that your organisation needs to consider a better way to work. Then you could be like Jon Stegner [Heath10], who wanted to change his manufacturing organisation's purchasing approach. Rather than write a presentation on how purchasing could be improved, he instead looked for evidence of the problem and simply presented his management team with a table containing a huge pile of the 424 different types of glove that various departments purchased, each tagged with its price ranging from $5 to $17. Such an immediate and visual demonstration of inefficiency convinced them change was needed in a heartbeat.

Conclusion

In this article, I've briefly outlined what I see as some problems in today's software development and explained at a high-level my understanding of how Systems Thinking can help. I've not had time here to go into any great detail but I hope you've seen enough to at least start considering the possibility of this approach. I believe that Systems Thinking has the potential to change the landscape, and I urge you to find out more and explore. My knowledge of this field is far from complete and I'd welcome any opportunity to hear about your explorations and build a body of experience and evidence to share with others.

Wishing you well with your adventures in software,

Tom. ■

References

[Deming82] Deming, W. Edwards (1982) *Out of the Crisis*.Cambridge, Mass: MIT Press.

[Heath10] Heath, Chip & Dan (2010) *Switch: How to change things when change is hard*. New York: Random House.

[Kelly11] Alan Kelly wrote about a framework for agile requirements management in *Overload* 101, in his article 'The Agile 10 Steps Model'.

[Seddon] John Seddon's organisation Vanguard Consulting Ltd has an authoritative website on systems thinking containing many useful resources and references to further reading: www.systemsthinking.co.uk

[Seddon03] Seddon, John (2003) *Freedom from Command and Control*. Buckingham, UK: Moreton Press.

The ACCU 2011 Crypto Challenge

Ready for the biggest challenge yet?
Richard Harris throws down the gauntlet.

For a third time I have had the pleasure of contributing a cryptographic puzzle to the fund raising efforts for Bletchley Park during the ACCU conference.

You may recall that the first two puzzles reflected the development of modern cryptosystems, beginning with electro-mechanical rotary ciphers such as the Enigma machine and followed by electronic stream ciphers such as RC4, and were designed to so that they could be broken with pencil and paper alone if their weaknesses were spotted.

As with the previous two puzzles a bonus question was included that was not possible to answer if the puzzle was solved by brute force rather than analysis.

So that you too might have a go at it, I present the puzzle below followed by its historical inspiration, its solution and the names of the conference delegates who cracked it.

The challenge

Encoding

The enemy are using a 32 character alphabet encoded as 5 bit signed binary numbers ranging from -16 to +15. The # character is a control code indicating that the following character should be interpreted as its numerical value rather than as a letter or punctuation. The full table of character mappings is given below.

```
---------------- +++++++++++++++
11111110000000000000000000111111
65432109876543210123456789012345
#abcdefghijklmnopqrstuvwxyz ?!.,
```

Encryption

The enemy are using a public key cryptosystem in which the private key is a set of variables containing the numeric encodings of randomly chosen characters.

The public key is a set of randomly generated polynomials in those variables, all of which, when calculated with the values from the private key, yield zero.

To transmit a message character, they first generate a new random polynomial for each of those in the public key. They multiply each pair and add the products together to create an encryption polynomial that must also yield zero when evaluated with the private key.

They finally add the encoded message character to yield the ciphertext polynomial.

To decrypt the ciphertext polynomial they simply evaluate it with the values in the private key; since the encryption polynomial will have a value of zero, the ciphertext polynomial will have a value of the encoded message character.

Example

The characters ‘r’ and ‘s’ give the private key variables x and y values of 2 and 3 respectively. The randomly generated polynomials x^2y+y and xy^2 yield 15 and 18 respectively and hence we can choose the polynomials x^2y+y-15 and xy^2-18 as the public key.

Multiplying these by the randomly generated polynomials y+1 and x-1 respectively and summing yields an encryption polynomial of

$$\begin{aligned} & (x^2y + y - 15) \times (y + 1) + (xy^2 - 18) \times (x - 1) \\ = \; & (x^2y^2 + y^2 - 15y) + (x^2y + y - 15) + (x^2y^2 - 18x) - (xy^2 - 18) \\ = \; & 2x^2y^2 + x^2y - xy^2 + y^2 - 18x - 14y + 3 \end{aligned}$$

The character ‘l’ can be encrypted by adding -4 to yield a ciphertext polynomial of

$$2x^2y^2 + x^2y - xy^2 + y^2 - 18x - 14y - 1$$

Evaluating this ciphertext polynomial with the private key values for x and y reveals the message.

$$\begin{aligned} & 2 \times 2^2 \times 3^2 + 2^2 \times 3 - 2 + 3^2 + 3^2 - 18 \times 2 - 14 \times 3 - 1 \\ = \; & 72 + 12 - 18 + 9 - 36 - 42 - 1 \\ = \; & 93 - 97 \\ = \; & -4 \\ = \; & \text{'l'} \end{aligned}$$

Public key

The enemy’s public key equations are

$$25w^2 + 9z^2 + 30wz - 49$$

$$9w^4 + 102w^2z - 30w^2 + 289z^2 - 170z + 25$$

$$7v^4 - 12v^2yz - 4y^2z^2 - 24v^2x + 4v^2 + 9x^2 - 30x - 24yz - 11$$

$$\begin{aligned} & 27v^2 - 50w^2 - 32x^2 + 147y^2 + 126vy + 80wx + 54v \\ & \qquad - 160w + 128x + 126y - 101 \end{aligned}$$

Bonus question

Assuming the worst possible ordering that is logically consistent with a cryptanalysis, what is the minimum necessary number of trial values that must be fed into these public key equations, as opposed to known values derived from them, to ensure that we can successfully recover the enemy’s private key values?

The historical justification

Historically, one of the biggest problems in cryptography was exchanging the secret keys needed to encrypt and decrypt messages.

Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

the first party provides a means for the second to **hide a secret** without providing the means to reveal it

For example, the development of effective electronic ciphers required the exchange of sequences of random numbers. One way to do this was to hire a courier to carry a tape containing such a sequence between one party and another, perhaps in an aluminium briefcase handcuffed to his wrist as seen in so many spy movies.

The problem was then how to securely exchange the keys to the briefcase.

A solution is for the first party to courier an *unlocked* briefcase to the second party *without* the key. The latter can put their tape into it, close and push a button to lock it, and then send the courier back to the first party to securely exchange their electronic key without ever having seen the briefcase key.

Of course, both parties must trust the courier…

This is essentially the idea behind public key cryptography; the first party provides a means for the second to hide a secret without providing the means to reveal it.

We do this by means of a trap door function, that is to say a function that is easy to compute but difficult to invert.

The canonical example is the factorisation of integers. It is easy to calculate the product of a pair of integer factors, but it is very much more difficult to determine from the product what those factors are.

Whilst this is certainly an interesting property of the integers, it is not at all clear on the face of it how it is of any use.

The trick is to construct a function which depends upon the product and is difficult to invert *unless* you know its factors.

The RSA cryptosystem is an example of just such a function.

The first step is to choose a pair of prime numbers p and q and compute their product n. For example

$$\begin{aligned} p &= 15 \\ q &= 11 \\ n &= 5 \times 11 = 55 \end{aligned}$$

Next we calculate the product of p-1 and q-1, denoted by $\varphi(n)$

$$\varphi(n) = (5-1) \times (11-1) = 4 \times 10 = 40$$

Now we must choose a value e that is both smaller than $\varphi(n)$ and has no common factors with it. In our example $\varphi(n)$ has prime factors of 2 and 5, so we may choose

$$e = 3 \times 3 \times 3 = 27$$

Next we must calculate the value d which when multiplied by e yields a product that is one plus some integer multiple of $\varphi(n)$

$$d \times e = 1 + k \times \varphi(n)$$

Mathematically, this the multiplicative inverse of e under arithmetic modulo $\varphi(n)$ which we can write as

$$d = e^{-1} \bmod \varphi(n)$$

We can do this using the extended Euclidean algorithm which we shall not cover here. Note that this inverse is guaranteed to be unique because e and $\varphi(n)$ have no common factors and in our example must be equal to 3.

$$3 \times 27 = 81 = 1 + 2 \times 40$$

The public key is comprised of the pair of integers n and e and the private key of the single integer d.

To encrypt a message m that has a value between 0 and n we calculate

$$c = m^e \bmod n$$

For example, we shall choose 2 for our message m. The ciphertext c is consequently

$$\begin{aligned} c &= 2^{27} \bmod 55 \\ &= 134217728 \bmod 55 \\ &= (2440322 \times 55 + 18) \bmod 55 \\ &= 18 \end{aligned}$$

To decrypt the ciphertext we compute

$$m = c^d \bmod n$$

In our example we have

$$\begin{aligned} m &= 18^3 \bmod 55 \\ &= 5832 \bmod 55 \\ &= (106 \times 55 + 2) \bmod 55 \\ &= 2 \end{aligned}$$

and as if by magic we have recovered the message!

What makes RSA secure is that the private key d is the multiplicative inverse of the public key value e modulo $\varphi(n)$ which in turn depends upon the hard to find prime factors of the public key value n.

We can demonstrate *why* RSA works from Euler's theorem which states that if m and n have no common factors, or in this case that the message is not equal to either of the prime factors of n, then

$$m^{\varphi(n)} \bmod n = 1$$

Consequently

$$\begin{aligned} c^d \bmod n &= (m^e)^d \bmod n \\ &= m^{e \times d} \bmod n \\ &= m^{1 + k \times \varphi(n)} \bmod n \\ &= m \times m^{k \times \varphi(n)} \bmod n \\ &= m \times (m^{\varphi(n)})^k \bmod n \\ &= m \times 1^k \bmod n \\ &= m \end{aligned}$$

since m is smaller than n.

The analysis of this first equation is a **significant clue** as to how we should proceed

The requirement that the message not be equal to either of the prime factors is not particularly burdensome in practice since they will be very, *very* large.

Now, our cryptosystem uses the difficulty in factoring integer polynomials rather than integers as its trap-door function. In the worked example we saw that it was relatively easy to construct a ciphertext polynomial from the public key polynomials.

That it is very much more difficult to reverse this process can be seen if you try to extract the message using just the ciphertext and public key polynomials.

$$x^2y + y - 15$$

$$xy^2 - 18$$

$$2x^2y^2 + x^2y - xy^2 + y^2 - 18x - 14y + 3$$

Integer polynomials like these are known as Diophantine equations and are notoriously difficult to reason about.

Perhaps the most famous example is Fermat's Last Theorem which states that the formula

$$a^n + b^n = c^n$$

has no solutions with integer a, b and c if n is greater than two.

Despite Fermat's claim to have a proof that was too large to fit in the margin of the text in which he wrote his conjecture, it took over 350 years before the theorem was finally, and famously, proven by Andrew Wiles.

The solution

In this challenge the ciphertext polynomial is not given, so it is 'simply' an exercise in finding the roots of the non-linear public key Diophantine equations; those values of the private key variables for which they all equate to zero.

Brace yourself…

To begin, consider the first public key equation

$$25w^2 + 9z^2 + 30wz - 49 = 0$$

Three of these terms are suspiciously perfect squares, $25w^2$, $9z^2$ and 49, and if we factorise the terms we have

$$5^2w^2 + 3^2z^2 + (2 \times 3 \times 5)wz - 7^2 = 0$$

The first three terms are those we should expect from the square of a sum of two terms and the last is the negation of a perfect square. Expressing the first three as just such a square and moving the last over to the right hand side yields

$$(5w + 3z)^2 = 7^2$$

We can now take the square roots of both sides of the equation to reveal two possible relationships between w and z.

The analysis of this first equation is a significant clue as to how we should proceed; the entire puzzle is in fact an exercise in solving quadratic equations and the key to solving it is realising this.

Examining the second equation

$$9w^4 + 102w^2z - 30w^2 + 289z^2 - 170z + 25 = 0$$

we should note that once again we have three perfect squares in the terms $9w^4$, $289z^2$ and 25.

Factorising all of the terms yields

$$\begin{aligned} 3^2w^4 &+ (2 \times 3 \times 17)w^2z - (2 \times 3 \times 5)w^2 \\ &+ 17^2z^2 - (2 \times 5 \times 17)z + 5^2 = 0 \end{aligned}$$

This time we have three non-squares that just happen to be twice the product of two of the three squared terms. Noting that it is only the terms with a single factor of five that are negative, we can express this equation as the square

$$(3w^2 + 17z - 5)^2 = 0$$

and hence

$$3w^2 + 17z - 5 = 0$$

We now have two possible pairs of simultaneous equations in w and z

$$\begin{aligned} 5w + 3z &= 7 \\ 3w^2 + 17z - 5 &= 0 \end{aligned}$$

and

$$\begin{aligned} 5w + 3z &= -7 \\ 3w^2 + 17z - 5 &= 0 \end{aligned}$$

and we are consequently in a position to examine the possible values of those private key variables.

Beginning with the first, we have

$$\begin{aligned} z &= (7 - 5w) \div 3 \\ 3w^2 + 17z - 5 &= 0 \end{aligned}$$

Multiplying the second equation by three, substituting z from the first and simplifying

$$\begin{aligned} 9w^2 + (3 \times 17)z - 15 &= 0 \\ 9w^2 + 17 \times (7 - 5w) - 15 &= 0 \\ 9w^2 + 17 \times 7 - 85w - 15 &= 0 \\ 9w^2 - 85w + 104 &= 0 \end{aligned}$$

At this point we can apply the rule for solving quadratic equations that we learnt in secondary school; minus a plus or minus the square root of b squared minus four times a times c all divided by two times a

We therefore have one candidate value but we must solve the second pair of equations to be sure that it is unique

$$ax^2 + bx + c = 0 \rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In this case this means that

$$\begin{aligned} w &= \frac{85 \pm \sqrt{85^2 - 4 \times 9 \times 104}}{2 \times 9} \\ &= \frac{85 \pm \sqrt{7225 - 3744}}{18} \\ &= \frac{85 \pm \sqrt{3481}}{18} \\ &= \frac{85 \pm 59}{18} \\ &= \frac{144}{18} \quad \text{or} \quad \frac{26}{18} \\ &= 8 \quad \text{or} \quad 1\frac{4}{9} \end{aligned}$$

We therefore have one candidate value for *w* of eight, but we must solve the second pair of equations to be sure that it is unique. Rearranging the first of those equations yields

$$z = (-7 - 5w) \div 3$$

$$3w^2 + 17z - 5 = 0$$

Following the same approach as before we have

$$\begin{aligned} 9w^2 + (3 \times 17)z - 15 &= 0 \\ 9w^2 + 17 \times (-7 - 5w) - 15 &= 0 \\ 9w^2 - 119 - 85w - 15 &= 0 \\ 9w^2 - 85w - 134 &= 0 \end{aligned}$$

Applying the rule for solving quadratic equations yields

$$\begin{aligned} \mathrm{w} &= \frac{85 \pm \sqrt{85^2 - 4 \times 9 \times -134}}{2 \times 9} \\ &= \frac{85 \pm \sqrt{7225 + 4824}}{18} \\ &= \frac{85 \pm \sqrt{12049}}{18} \end{aligned}$$

We must stop here because 12049 is a prime and hence this equation cannot result in a valid value for *w*.

With the unique valid value we have for *w* we can now calculate the correct value for *z*

$$\begin{aligned} w &= 8 \\ z &= (7 - 5w) \div 3 \\ &= (7 - 40) \div 3 \\ &= -33 \div 3 \\ &= -11 \end{aligned}$$

Having solved the first two public key equations we are ready to analyse the third

$$7v^4 - 12v^2yz - 4y^2z^2 - 24v^2x + 4v^2 + 9x^2 - 30x - 24yz - 11 = 0$$

Ignoring the coefficients for now, this equation has a constant term, terms in v^2, *x* and *yz*, their squares and terms in all of their pair products except *xyz*. The fact that this term is missing mean that this cannot be a square of a sum of terms in v^2, *x* and *yz*.

It is, however, perfectly consistent with a sum of squares of the form

$$(av^2 + byz + c)^2 - (dv^2 + ex + f)^2 = 0$$

Noting that the v^4, v^2 and constant terms must be split between both squares, we can group the related terms together on either side of the equals sign

$$7v^4 - 12v^2yz - 4y^2z^2 + 4v^2 - 24yz - 11 = 24v^2x - 9x^2 + 30x$$

We can see immediately that the y^2z^2 and x^2 terms on the left and right hand side of the equation have negative coefficients. We must therefore negate both sides

$$-7v^4 + 12v^2yz + 4y^2z^2 - 4v^2 + 24yz + 11 = -24v^2x + 9x^2 - 30x$$

Now we must add the terms that are missing on the right hand side of the equation to both sides

$$\begin{aligned} &(a - 7)v^4 + 12v^2yz + 4y^2z^2 + (b - 4)v^2 + 24yz + (11 + c) \\ &\qquad = av^4 - 24v^2x + bv^2 + 9x^2 - 30x + c \end{aligned}$$

Now, if both sides are squares of sums, these unknown coefficients must conform to the very specific relationships this implies.

Firstly, the constant *c* must be related to the *x* and the x^2 coefficients by

$$\begin{aligned} -30 &= 2 \times \sqrt{9} \times \sqrt{c} \\ c &= 25 \end{aligned}$$

giving

$$\begin{aligned} &(a - 7)v^4 + 12v^2yz + 4y^2z^2 + (b - 4)v^2 + 24yz + 36 \\ &\qquad = av^4 - 24v^2x + bv^2 + 9x^2 - 30x + 25 \end{aligned}$$

reassuringly making the constants on both sides of the equation perfect squares.

Similarly, the coefficient of v^4 on the right hand side, *a*, must be related to the coefficients of v^2x and x^2 by

$$\begin{aligned} -24 &= 2 \times \sqrt{9} \times \sqrt{a} \\ a &= 16 \end{aligned}$$

A problem immediately presents itself in that the coefficients of the squared variables are not squares

giving

$$\begin{aligned}9v^2 + 12v^2yz + 4y^2z^2 + (b-4)v^2 + 24yz + 36 \\ = 16v^4 - 24v^2x + bv^2 + 9x^2 - 30x + 25\end{aligned}$$

once again yielding perfect squares exactly where we want them!

Finally, relating the v^2 coefficient on the right hand side, b, to those of v^4 and the constant means that

$$\begin{aligned}b &= 2 \times \sqrt{16} \times \sqrt{25} \\ b &= \pm 40\end{aligned}$$

Given that the coefficients of both x and v^2x are negative, b must be positive giving

$$\begin{aligned}9v^4 + 12v^2yz + 4y^2z^2 + 36v^2 + 24yz + 36 \\ = 16v^4 - 24v^2x + 40v^2 + 9x^2 - 30x + 25\end{aligned}$$

Partially factoring the coefficients yields

$$\begin{aligned}3^2v^4 + (2 \times 2 \times 3)v^2yz + 2^2y^2z^2 + 6^2v^2 + (2 \times 2 \times 6)yz + 6^2 \\ = 4^2v^2 - (2 \times 3 \times 4)v^2x + (2 \times 4 \times 5)v^2 + 3^2x^2 - (2 \times 3 \times 5)x + 5^2\end{aligned}$$

so we can express this as

$$(3v^2 + 2yz + 6)^2 = (4v^2 - 3x + 5)^2$$

and consequently

$$3v^2 + 2yz + 6 = \pm(4v^2 - 3x + 5)$$

Now we have already determined that z is equal to minus 11, so we can substitute it into this equation to yield

$$3v^2 - 22y + 6 = \pm(4v^2 - 3x + 5)$$

Now we are on the home stretch with just one public key equation left to consider

$$\begin{aligned}27v^2 - 50w^2 - 32x^2 + 147y^2 + 126vy + 80wx + 54v - 160w \\ + 128x + 126y - 101 = 0\end{aligned}$$

Leaving aside the coefficients once again, it is clear that this formula is also formed from two quadratic equations, one in v and y, the other in w and x, with the constant term split between them.

Rearranging gives

$$\begin{aligned}27v^2 + 147y^2 + 126vy + 54v + 126y + a - 101 \\ = 50w^2 + 32x^2 - 80wx + 160w - 128x + a\end{aligned}$$

A problem immediately presents itself in that the coefficients of the squared variables are not squares.

To address this we must partially factor the coefficients *first*.

$$\begin{aligned}&(3 \times 3^2)v^2 + (3 \times 7^2)y^2 + (3 \times 2 \times 3 \times 7)vy \\ &+ (3 \times 2 \times 3 \times 3)v + (3 \times 2 \times 3 \times 7)y + a - 101 \\ &= (2 \times 5^2)w^2 + (2 \times 4^2)x^2 - (2 \times 2 \times 5 \times 4)wx \\ &\quad + (2 \times 2 \times 5 \times 8)w - (2 \times 2 \times 4 \times 8)x + a\end{aligned}$$

So far, this is consistent with the left hand side being three times a square of a sum and the right hand side being twice the square of a sum.

If we take these factors out, we shall see whether it gives a consistent value for the constant a.

$$\begin{aligned}&3\Big[3^2v^2 + 7^2y^2 + (2 \times 3 \times 7)vy + (2 \times 3 \times 3)v \\ &\qquad + (2 \times 3 \times 7)y + \tfrac{1}{3}(a - 101)\Big] \\ &= 2\Big[5^2w^2 + 4^2x^2 - (2 \times 5 \times 4)wx + (2 \times 5 \times 8)w \\ &\qquad - (2 \times 4 \times 8)x + \tfrac{1}{2}a\Big]\end{aligned}$$

From the right hand side, following a similar argument that we used for the previous formula, if it is a square then a must satisfy

$$\begin{aligned}2 \times 5 \times 8 &= 2 \times \sqrt{5^2} \times \sqrt{\tfrac{1}{2}a} \\ a &= 128\end{aligned}$$

and therefore

$$\begin{aligned}3[3^2v^2 + 7^2y^2 + (2 \times 3 \times 7)vy + (2 \times 3 \times 3)v + (2 \times 3 \times 7)y + 9] \\ = 2[5^2w^2 + 4^2x^2 - (2 \times 5 \times 4)wx + (2 \times 5 \times 8)w - (2 \times 4 \times 8)x + 64]\end{aligned}$$

giving us exactly the squares we seek

$$3(3v + 7y + 3)^2 = 2(5w - 4x + 8)^2$$

Given that the private key variables are integers and that the square root of three is not an integer multiple of that of two, this equation can only hold when both the left and right hand sides are equal to zero

$$\begin{aligned}3v + 7y + 3 &= 0 \\ 5w - 4x + 8 &= 0\end{aligned}$$

We already know that w is eight, so from the second equation we have

$$\begin{aligned}5 \times 8 - 4x + 8 &= 0 \\ -4x &= -48 \\ x &= 12\end{aligned}$$

Having found x we can further simplify the equation we derived from the third public key equation

$$3v^2 - 22y + 6 = \pm(4v^2 - 3x + 5)$$

$$3v^2 - 22y + 6 = \pm(4v^2 - 31)$$

From the first of the equations we derived from the fourth we can relate v and y

And after all, it was an **exercise in public key cryptography**; of course it was going to be difficult!

$$v = -(7y+3) \div 3$$

Substituting this into the above is more easily achieved if we multiply the latter by nine

$$\begin{aligned} 27v^2 - 198y + 54 &= \pm(36v^2 - 279) \\ 3(7y+3)^2 - 198y + 54 &= \pm(4(7y+3)^2 - 279) \end{aligned}$$

Expanding the squares yields

$$\begin{aligned} 3(7y+3)^2 - 198y + 54 &= \pm(4(7y+3)^2 - 279) \\ 3(49y^2 + 42y + 9) - 198y + 54 &= \pm(4(49y^2 + 42y + 9) - 279) \\ (147y^2 + 126y + 27) - 198y + 54 &= \pm((196y^2 + 168y + 36) - 279) \\ 147y^2 - 72y + 81 &= \pm(196y^2 + 168y + 243) \end{aligned}$$

Once again we have two possibilities to consider, both of which yield simple quadratic equations.

Firstly

$$\begin{aligned} 147y^2 - 72y + 81 &= +(196y^2 + 168y - 243) \\ 49y^2 + 240y - 324 &= 0 \end{aligned}$$

Using our trusty quadratic equation rule again we have

$$\begin{aligned} y &= \frac{-240 \pm \sqrt{240^2 - 4 \times 49 \times -324}}{2 \times 49} \\ &= \frac{-240 \pm \sqrt{57600 + 63504}}{98} \\ &= \frac{-240 \pm \sqrt{121104}}{98} \\ &= \frac{-240 \pm 384}{98} \\ &= \frac{108}{98} \quad \text{or} \quad \frac{-588}{98} \\ &= 1\frac{5}{49} \quad \text{or} \quad -6 \end{aligned}$$

giving us a candidate value for y.

Of course we must check the second possibility to be sure we have the correct value

$$\begin{aligned} 147y^2 - 72y + 81 &= -(196y^2 + 168y - 243) \\ 343y^2 + 96y - 162 &= 0 \end{aligned}$$

Solving for y

$$\begin{aligned} y &= \frac{-96 \pm \sqrt{96^2 - 4 \times 343 \times -162}}{2 \times 343} \\ &= \frac{-96 \pm \sqrt{9216 + 222264}}{686} \\ &= \frac{-96 \pm \sqrt{231480}}{686} \\ &= \frac{-96 \pm \sqrt{2^3 \times 3^2 \times 5 \times 643}}{686} \end{aligned}$$

Consequently y must be equal to minus six and thus

$$\begin{aligned} v &= -(7y+3) \div 3 \\ &= -(-42+3) \div 3 \\ &= 39 \div 3 \\ &= 13 \end{aligned}$$

giving us the values of all of the private key variables without our having to guess at any of them!

An (insincere) apology

Now I recognise that this puzzle was somewhat trickier than the previous puzzles and for that I offer my apologies. In my defence, it was quite difficult designing a public key crypto challenge that satisfied my own requirements; those of novelty and a pencil and paper solution. Everything else I considered was either trivially solved using brute force or impossible to solve using pencil and paper.

And after all, it *was* an exercise in public key cryptography; of course it was going to be difficult!

I'm afraid that this has been the last of the crypto challenges; the next logical step would have been a puzzle based on quantum cryptography but I am reasonably certain that it is quite beyond my meagre faculties to design such a beast.

And finally...

Congratulations are due to Gary Duke who successfully cracked the Crypto Challenge and to Per Liboriussen who was just one step from doing so.

I should also like to thank everyone who took part and everyone who donated to Bletchley Park. ■