overload 102

APRIL 2011 £3

Benefits of Well-Known Interfaces in Closed Source Code

Sticking to widely-known interfaces can be a good idea, even in closed code.

On CMM, Formalism and Creativity

We investigate the relationship between formal development methods and creativity.

The Agile Spectrum

A look at development processes, and how they range on a "spectrum" of agility.

Refactoring and Software Complexity Variability

An interesting attempt to model software complexity and investigate how refactoring can affect it.

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 102

April 2011

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 103 should be submitted by 1st May 2011 and for Overload 104 by 1st July 2011.

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.

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

This Year's Model

Design and development requires us to think about the world. Ric Parkin considers some ways of coping with the complexity.

How do you predict the future? Lacking a time machine or a handy crystal ball, we have to resort to more mundane methods. These usually involve making a model, perhaps unconciously, that can answer usefully relevant questions. Sometimes these models can be of breathtaking simplicity: want to predict what the weather will be like tomorrow? Look out of the window – there's a good chance that it'll be the same as today! This not actually that bad a model – in many places the weather tends to change only slowly over time (and in many tropical climes stays pretty much constant over the whole year). Also there are often stable weather patterns which persist for many days, so making prediction easy – a high pressure over continental Europe can remain there for many days, stretching into weeks.

The weather-obsessed UK is actually one of the hardest to predict – it sits at the end of the northern jet stream, a high altitude band of fast winds which start over north Africa and encircle the globe moving slowly northward, across India, Japan, the US, and finally dissipating above the UK. But as this 'river in the sky' is buffetted around it moves to the north or south of the UK. As many Atlantic weather systems are guided by the jet stream, if it's sending them towards you then you know that the weather will be very changable with bands of rain followed by clear spells. If it's dumping them over Iceland while a European high pressure extends over the UK, you'll have very stable weather – hot clear weeks over summer. There – we've just extended our mental model to allow for even better predictions, by understanding some of the processes that affect the result we're interested in. To go further you might start writing proper mathematical or computer models of atmospheric circulations, to try and predict finer details of how and when these large scale features change.

So what's this got to do with software? Well, we use models a lot as well. Sometimes they're what we're programming, but most likely they're more subtle than that. One will be a model of people's intuition: if you're designing a user interface it is a good idea to understand how a user thinks about what they want to do, and how your interface will fit into that 'narrative'. A poor interface will cause them to come to a shuddering halt as they work out what they need to do; a good interface by contrast meshes well with their model and allows them to carry out their work with little impedement. A good interface should appear 'transparent' to the user – they just use it without conciously thinking.

One example I've had of this was when I was working on a program that had a very strong visual aspect – networks of information were represented by icons with links between them, and you could just pick up and move the icons. One problem came at the edges of the screen – we wanted an 'auto-scroll' feature to reveal more of the virtual sheet of paper. To make thing more complex you could drag from one window and drop into another, so the obvious solution of scrolling when you dragged outside didn't work. To start with I tried having a 'sensitive zone' just inside the window which would activate the scrolling. Unfortunately people found they couldn't control it. It would start scrolling when they were doing something else, or scroll too fast so they overshot they target, or scroll too slow so they were sitting there waiting. I got a *lot* of bug reports and many change requests for this suggesting all sorts of ideas, usually contradictory, and sometimes suggesting a complete rethink, or wanting many options to 'control' it (I'm of the opinion that many options are added that only give the appearance of control, to hide the problem of things not working properly – instead it transfers the problem of fixing it onto the user)! In this case I persevered, and using the many complaints as inputs as to what sort of things didn't work finally came up with a simple but effective solution – a delay before scolling started, long enough that it wouldn't get triggered accidentally; a fast outer and a slower inner sensitive zone, where scroll speed increased the further out you went, increasing gently to start with but quickly up to the edge. Suddenly people could control the scrolling, its simplicity was easy to predict, and very quickly it became automatic and all the change requests dried up – it had become invisible.

Another type of interface is an API used by programmers. These too should strive to mesh with the mental model of what an interface should do (and should not), otherwise confusion, frustration and bugs become the norm. Arun Saha's article in this issue deals with exactly this problem.

What other models do we use? Task time estimation is a very common one, but how do we do it? We don't just guess, we use our experiences to come up with a reasonable estimate based on various factors. A start would be a quick guess at how much work is involved, perhaps based on a comparision with a similar task we've done before. We can also make adjustments based on knowing how difficult it is to change code in the relevant area – a good example of this is date and time processing, which ought to be simple and yet we still see problems [BBC]. I have a theory as to why this particular example is so error prone – it seems superficially simple so people dive in, and yet when you look at the details needed for various applications there are many subtle complications, from calender changes (and countries changing at different times), time zones (including historical changes), summer time change rules (and exceptions), all the way down to taking into account leap seconds, the varying spin of the earth, and the time dilation due to General Relativistic effects!

One factor that isn't captured by a simple estimate is the spread of possible outcomes – 'two months' sounds definite, but in reality it'll normally be 'around two months, a week earlier if all goes well, but could be three months if we find problems. Four if they're bad'. It's hard to plan with that sort of uncertainty. But models can come to our rescue here – we could expect that as the worst case is really bad, but the best case is only a bit better, then on average the most likely time will be a bit worse than the

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.

simple estimate. If we use the expected time rather than the estimate time, we've taken into account some of the inevitable problems.

However this assumes that the chance of a problem in a task is independent of the probability of a problem in another task. While this may be true for relatively seperate tasks, quite often in code the tasks will be related in some way, perhaps by being in the same area of nasty buggy code. In which case they are no longer independent, and our model is going to be wrong, because if Task A is late, then the chances of Task B being late is more than we suspected, so we have been optimistic.

We do have some hope though – if we suspect a group of tasks are not independent, then we can use the actual time taken for some to adjust our estimates for the later ones. eg Task A's expected time was 1 month, but took a week longer. So assuming Task B is dependent on the same issues that caused that delay, we could adjust the estimate from 2 months to 2 weeks longer. This is very similar to Baysian Inference [Bayes] where you adjust a probability based on new information gained from a non-independent observation. (This is a strangely counter-intuative subject, but can be very powerful.)

I would be interested to hear if anyone has tried this sort of adjustment – I suspect it could work if Task A leaves the code with the same latent problems, however refactoring to leave it in a better state will reduce the adjustment needed. Perhaps an iterative adjustment is needed: in the light of Task A adjust Task B, and in the light of that adjust Task C and so on.

This shows one of the fundamental things to be aware of in models – they are a simplification of the world in order to make a prediction, but you should be aware of what you have simplified and assumed, and when those assumptions break down. In this issue we have an article by Alex Yakyma on an attempt to build a model of software complexity, and it generated quite a bit of discussion for many of these reasons – what were the model's assumptions and were they reasonable, were the factors really independent, etc. All good questions – it would be interesting to see what effects changing one of these assumptions would have? Or a more detailed look at how task estimates combine under various assumptions, or some other aspect of software development.

Anniversaries

Being a bi-monthly magazine, there are always a few notable anniversaries coming and going, but this issue has had a few interesting ones. There's been much in the media about the 5th anniversary of Twitter. That's not very long, and yet it has become quite pervasive. To take two recent examples – the recent 'arab spring' wave of demonstrations seem to have been organised on an ad hoc basis by ordinary people using modern decentralised communications. Even when the mainstream news were controlled by a government, people were reporting events themselves via mobile video, twitter updates and Facebook groups. The speed at which these events unfolded was remarkable, based in no small part on cheap fast mobile phones and computers.

There was also an uglier side to Twitter in the news as well. You may have heard of a 13 year old called Rebecca Black. She'd recorded a song and video [Black] which went viral and has had (at the last count) 66.9 million views on Youtube (whose 6th anniversary is in April [Youtube]) and over 1.1 million comments, and was a top trend on Twitter. Unfortunately a lot of the reaction was not just negative, but downright nasty. I won't comment on the song, but it seems the speed of modern commenting plus the ability to be anonymous (or just in a crowd) can bring out the vicious side of some people. This is not a new phenomenon either – there's been flame wars on email and newsgroups since they were invented, and with 'fast reaction' communication like Twitter it's even easier to fire off an ill-thought through, or even nasty, message. Perhaps a return to 'slow' communication would help? There's been an add in for gmail for some time now that forces you to answer some simple sums before it'll send a message, on the theory that if you're tired and/or drunk enough to fail, you'll probably regret the email [Gmail]

More important to many of us I suspect, we have just passed the 30th anniversary of the Sinclair ZX81 [ZX81]. This was the time when many people were getting their first glimpse of home computing, even if most didn't know what to do with it! But there were many who didn't care, and just loved playing with getting this funny black box to do strange things. With only 1K of RAM, which was used for data, code, and also for video memory, applications were limited (you could get a 'Ram pack' to extend memory by a whole 16K, but these were notoriously wobbly. Some swore by blu-tac, I found a fabric plaster stabilised it enough). But that curse was also a blessing – it forced people to be extremely clever at finding neat ways of getting the most out of it, which some have suggested led to the UK having such a lot of ingenious programmers.

Mobile phones are even older – the first call was made on 3rd April 1973 by Martin Cooper [Cooper], who was leading the research team at Motorola to buld one. Who did he call? His rival at AT&T to tell him he'd got one working first.

References

[Bayes] http://en.wikipedia.org/wiki/Bayesian_statistics

[BBC] http://www.bbc.co.uk/news/technology-12104890 and http://www.bbc.co.uk/news/technology-12878517

[Black] http://en.wikipedia.org/wiki/Friday_(Rebecca_Black_song)

[Cooper] http://inventors.about.com/cs/inventorsalphabet/a/martin_cooper.htm

[Gmail] http://gmailblog.blogspot.com/2008/10/new-in-labs-stop-sending-mail-you-later.html

[Youtube] First ever video: http://www.youtube.com/watch?v=jNQXAC9IVRw

[ZX81] http://en.wikipedia.org/wiki/ZX81

Benefits of Well Known Interfaces in Closed Source Code

Designing a good API is a significant challenge. Arun Saha suggests taking inspiration from outside.

The availability of a high quality data structure library is a necessary ingredient for the success and timely completion of any software project. It allows the programmers to focus on the problem domain rather than the solution domain. But what are the options if no such library is available and an in-house one has to be developed? Fortunately, all is not lost. The in-house library can be designed to use a standardized or well-known interface, which reduces a lot of the strategic design, tactical design, testing, learning, adaptation, and maintenance efforts. This article focuses on two key aspects, interface design and functional testing.

Introduction

Consistent use of a library keeps uniformity, both syntactic and semantic, across a project. It is essential for the development and maintenance of any large or multi-programmer code base. In C++, the standard library specifies a bunch of data structures (a.k.a. containers) (for example, `array`, `vector`, `list`, `map`, `set`, `unordered_map`, `unordered_set`, `bitset`) and algorithms (for example, `find`, `search`, `sort`, `partial_sort`) that are usable with *any* suitable built-in or user-defined type [C++2011, relevant sections: 20, 23, 24, 25]. The availability of the standard library provides immense benefits to a project: the programmers can look beyond the repetitive structural and algorithmic issues and focus more on the issues of the problem domain. The first implementation of such a type independent library was published by SGI and is known as Standard Template Library.

Although these containers and algorithms are specified in the C++ standards (C++1998, C++2003, and upcoming C++0x), they are not part of the core C++ *language*; the library extends the language to provide some general components [Josuttis99].

There are multiple implementations of the C++ standard library available. Among them, SGI, GNU and STLport are open-source implementations, and Dinkumware is a commercial one. [Implementations]

However, there exist systems and environments, mostly embedded systems, where the C++ language is used without the standard library. One such example is 'Embedded C++' [EC++]; it is a subset of C++ which prohibits templates (among other things) and thereby a major part of standard library, including the containers and the algorithms, is unavailable.

If some project wants to use the standard library and if one of the open-source implementations is technically and legally suitable, then that can be chosen to be used – end of story.

Arun Saha is a senior member of technical staff at Fujitsu Network Communications, Sunnyvale. He works on security and accuracy in wireless localisation and carrier ethernet switching. He is the author of *Secure Protocols for Location, Adjacency, and Identity Verification*. He can be reached at: saha@cs.ucr.edu

However, in a commercial software or a proprietary code base, using open-source software is frequently not an option. There are multiple reasons, and the following is a non-exhaustive list:

- licensing or legal issues (for example, the requirement of publishing derivative work or modifications to the open-source code)
- the code is not actively maintained (for example, as of March 2011, the latest release of STLport is from December 2008)
- the code is not well documented and hence difficult to understand and maintain
- the code does not match the in-house development policies or coding standards (for example, the use of exceptions or asserts) and changing them requires significant rework.

Thus, the commercial houses have two major options for using a C++ data structure library:

- Option A : Purchase the library software from a vendor and license it appropriately
- Option B : Develop the necessary library *in-house*.

Our experience is with Option B (Develop), and in the remainder of this article we shall share two major lessons learned from that choice. One is the interface design and the other is comparative testing.

Interface design

The first and foremost item in developing a library is designing the interface. By interface, we mean all the public methods and attributes that are visible to the user code. While it is possible to design an interface in multiple ways, it is hard to produce the 'right' one. However, though the choice of Option B means developing an in-house implementation, fortunately there is still something that can be 'borrowed' from the C++ standard library. The interface!

For the interface of the to-be-developed library, our recommendation is to choose *exactly* the one specified in the C++ standard.

There are many reasons why.

It is *the* standard

API design is hard. A study of the obstacles faced by developers when learning APIs [Robillard09] notes:

> APIs support code reuse, provide high-level abstractions that facilitate programming tasks, and help unify the programming experience (for example, by providing a uniform way to interact with list structures).

The interface of the C++ standard library is widely known; virtually every C++ programmer is aware of it. For example, to insert an item at the end of a `list` or `vector`, the de facto, the idiomatic, and the most natural way is to use the method `push_back()`.

The opinions expressed in this article are solely the author's – not his employer's.

It would be a bigger pity if programmers have to, on top of that, learn different APIs for doing the same job

Since it is *the* standard, some other benefits include:

- **Known Roadmap:** Between the library developers and the library users, there is a clear understanding about what is offered or may be offered versus what is not.
- **Reference point:** In case of any confusion or disagreement internal to the library development team or between library developers and library users, the standard specification serves as the authoritative reference point.
- **Time savings:** Following a standard eliminates the design debates and the time spent on interface design.
- **Superior Design:** The API of the C++ standard library is standardized by the C++ standardization committee which includes many of the world's top C++ experts. Over the time, it has also been reviewed by other experts outside the committee and used by thousands of projects by millions of users. As a result of such rigorous analysis, extensive review, and widespread use, the interface has become so robust that it would be short sighted to ignore it.
- **Cultural effect:** The users of the library have the *feel* of using the C++ standard library, albeit an in-house implementation.

Lower barrier to entry

One of the costs (and often a barrier) of using a library is learning its interface. The aforementioned study warns that:

> APIs have grown very large and diverse, which has prompted some to question their usability. It would be a pity if the difficulty of using APIs would nullify the productivity gains they offer.

It would be a bigger pity if programmers have to, on top of that, learn *different* APIs – for example the C++ standard library and potentially different in-house libraries at different organizations – for doing the same job, such as inserting an element to a `list`. The number of APIs that we are talking here is large: dozens of classes, each with scores of methods, scores of algorithms, and a long list of idioms and good practices. There exists a significant amount of material – books, articles, tutorials, blogs, forums, newsgroups, mailing lists – on aspects of the C++ standard library; it is a substantial learning curve to master them and become an effective user.

If the in-house library uses the same API as the C++ standard library, then the cost of training the programmers is completely eliminated (or drastically reduced) because they can simply continue to apply their pre-acquired knowledge (or learn from already existing materials). This applies equally well for the C++-skilled programmers who are hired in future. On the contrary, if the in-house library is built with a different API, all the knowledge and mastery suddenly becomes useless.

Long term impact

Any software interface, standardized or otherwise, has long term implications. The implementation can be easily modified, but once it is published and the remaining code base starts using it, changing an interface is extremely hard. Choosing an already stable interface reduces such impacts.

Also, if for some reason, in future, the organization wants to switch from Option B (Develop) to Option A (Purchase), then the migration is extremely easy because all user code is written against the same interface.

Rule of least surprise

The Art of Unix Programming [Raymond03] observes:

> The easiest programs to use are those that demand the least new learning from the user – or, to put it another way, the easiest programs to use are those that most effectively connect to the user's pre-existing knowledge.

So, following an *existing* standard is the most natural choice to make.

Testability

If the in-house library follows the same interface as the C++ standard library, then testing the correctness of the library is much easier. This important aspect is now explained in more detail.

Testing

The choice of interface specification is a good first step, but that itself is not sufficient. The crucial design invariant – the interface compatibility with the C++ standard library – has to be actively maintained. That leads to the following questions:

- **Syntax conformance** Does the in-house library conform to the interface specified by the C++ standard library?
- **Semantic conformance** Does the in-house library provide behaviour exactly as specified in the C++ standard library?

The solution that we found most useful is to develop a test suite for the library with the following strategy:

1. Each unit of the library, for example a container, an iterator, an algorithm, or an allocator has its own unit test.
2. Separate unit tests are independent and stand-alone C++ programs, all of which are run in a regression suite.
3. The unit tests verifies the behaviour of a unit against the specification in the C++ standard.
4. A unit test exercises each interface of the unit in all possible ways.

It is best to explain with examples. In the following, excerpts from the `vector` test code are shown.

Comparative testing

All the tests follow a common structure: at the beginning of the test code, a control is provided to run the test against either a reference standard, or the in-house code. Listing 1 shows the structure for `vector`.

First it defines the type of the elements that the `vector` consists of. For simplicity in this example, we used the built-in type `unsigned long int`, although it could be any user defined type (`struct` or `class`). When the macro `STD_REF` is defined, we run this unit test on a reference

Overall, it has proven to be a great step in reducing software complexity in the organization's code base

implementation of the standard library. Otherwise, we run this unit test on the in-house library. Observe that, in both ways of setup, we defined a type named **`TypeVector`**. The remainder of the file `vector_test.cpp` runs all tests on **`TypeVector`**, *without any knowledge of the source of the library code*.

```
// vector_test.cpp
typedef unsigned long int Type;
#ifdef STD_REF
  #include <vector>       // From standard library
  typedef std::vector< Type > TypeVector;
#else
  #include "vector.hh"    // From in-house library
  typedef inhouse::vector< Type > TypeVector;
#endif
typedef TypeVector iterator TypeVectorIter;

#include <cassert>
#define UNIT_TEST assert

static const Type Values[] = {10, 20, 30, 40, 50,
   60, 70};
static const size_t ValuesLength =
   sizeof( Values ) / sizeof( Values[ 0 ] );
int main() {
  size_t valuesIndex = 0;
  TypeVector vut;  // Vector Under Test

  UNIT_TEST( vut.empty() );
  for( valuesIndex = 0;
       valuesIndex < ValuesLength;
       ++valuesIndex ) {
    vut.push_back( Values[ valuesIndex ] );
  }

  UNIT_TEST( ! vut.empty() );
  UNIT_TEST( vut.size() == ValuesLength );
  UNIT_TEST( vut.front() == Values[ 0 ] );
  UNIT_TEST( vut.back() ==
     Values[ ValuesLength - 1 ] );
  valuesIndex = 0;
  for( TypeVectorIter it = vut.begin();
       it != vut.end();
       ++it, ++valuesIndex ) {
    UNIT_TEST( *it == Values[ valuesIndex ] );
    UNIT_TEST( *it == vut[ valuesIndex ] );
    UNIT_TEST( *it == vut.at( valuesIndex ) );
  }
  UNIT_TEST( valuesIndex == ValuesLength );
  UNIT_TEST( ! vut.empty() );
  vut.clear();
  UNIT_TEST( vut.empty() );
 }
```

Listing 1

Thus we have a simple way of choosing one among many possible **`vector`** implementations and run the unit test on the chosen one. If the implementations conform to the C++ standard, then the unit test would compile with all of them, and execute to produce identical results in all of them.

Test construction

The next task of the unit testing strategy is creating the test cases. All the test cases are created as a sequence of two steps:

1. Do some operation(s) on the unit (here, **`vector`**).
2. Programmatically verify that the properties and contents of the data structure matches the expected result(s).

The rest of Listing 1 shows an example of some simple test cases applied on **`vector`**, where programmatic verification is done using asserts.

It tests some methods of **`vector`** (**`empty()`**, **`push_back()`**, **`size()`**, **`front()`**, **`back()`**, **`at()`**, **`begin()`**, **`end()`**, **`clear()`**, **`operator[]`**) and the type **`vector::iterator`**.

For each unit, the conformance and correctness testing consists of few simple steps. The steps for compiling and running for **`vector`** are as follows:

1. **`CC := g++ -W -Wall -Werror -ansi -pedantic -std=c++0x`**
2. **`CC -DSTD_REF -D_GLIBCXX_DEBUG vector_test.cpp -o ref_vector`**
3. **`CC vector_test.cpp -o inhouse_vector`**
4. **`./ref_vector`**
5. **`./inhouse_vector`**

Things to note for these steps:

- **Build settings** The compiler used is the GNU C++ compiler with all the warnings turned on and strict conformance to the C++0x standard.

  For use in the target environment, the in-house library is (cross) compiled and linked with a different C++ compiler, and (successfully) run on a different OS on a different CPU.
- **Reference build** The unit test code is built to be run with the reference standard library. Also, the GNU STL debug macro is defined for strict checks. Successful completion of this step implies that the unit test code in `vector_test.cpp` is syntax compliant with the reference C++ standard library.
- **Inhouse build** The unit test code is built to be run with the in-house library. Successful completion of this step along with the previous step implies that the in-house library is also syntax compliant with the C++ standard.

- **Reference execution** The unit test is executed on the reference standard library. Successful completion implies that the unit testing code (`vector_test.cpp`) is semantically correct.
- **Inhouse execution** The unit test is executed on the in-house library. Successful completion proves that the in-house library is semantically compliant to the C++ standard. In other words, the in-house **`vector`** implementation exhibited expected standard behavior.

This example is rather simplistic, it uses only few member functions available in **`vector`**. In reality, there are lot more methods in the **`vector`** template class. To obtain basic confidence in the conformance and correctness of the in-house library, the unit test code tests *each* method in *isolation*. Then the methods are tested in different *combinations* and *sequences*.

Other experiences

Without risking any non-conformance to the standard interface, the implementation of the in-house library can offer some niceties which may or may not be available in other implementations. Here are two examples.

- Have log/trace messages at important points in the code that can be triggered based on a log level selected by the user code. For example, generation of log messages whenever memory is allocated or deallocated.
- Maintain class invariants. For example, in the **`list`** class template, we kept the following private attributes:
  - **`size_`**: number of elements in the **`list`** (this also helped the **`size()`** method to have O(1) complexity)
  - **`news_`**: number of times an element is added to the **`list`**.
  - **`deletes_`**: number of times an element is removed from the **`list`**.

  Thus we had the following invariant

  ```
  news_ - deletes_ == size_
  ```

  We asserted on this invariant as a pre-condition and post-condition of every mutator method in the **`list`** class template.

Some other general strategies:

- Writing the unit tests (for example `vector_test.cpp`) before implementing the unit (for example **`vector`**). Since it is known what exactly to expect, the development of a standards compliant in-house library is an ideal scenario for applying the principles of Test Driven Development [TDD], and employing it has been immensely helpful to us.
- Comparison of program size, for example comparing size between **`ref_vector`** and **`inhouse_vector`**
- Comparison of program speed, for example comparing running time between **`ref_vector`** and **`inhouse_vector`**

Conclusion

Consistent interfaces make life easier. The same is true for software development. This article emphasizes that the interface provided by the C++ standard library, which sometimes go unappreciated and overlooked,is very valuable by itself. As the author of the in-house library, it has been realized numerous times that choosing to follow the standard interface was the most important design decision that was made. Following the interface conventions as in the C++ standard library has tremendously helped (non-library) programmers to easily understand and easily use the newly written in-house library. It brought the programmers to a common and consistent style both syntactically and semantically. Overall, it has proven to be a great step in reducing software complexity in the organization's code base. ■

References

[C++2011] 'Working Draft, Standard for Programming Language C++', 02 2011. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf

[EC++] 'The Embedded C++ specification', 1999. http://www.caravan.net/ec2plus/spec.html

[Implementations]'Dinkumware C++ Standard Library'. (http://www.dinkumware.com/), 'The GNU C++ Library Documentation' (http://gcc.gnu.org/onlinedocs/libstdc++/), 'SGI Standard Template Library Programmer's Guide', 1994 (http://www.sgi.com/tech/stl/), 'STLport C++ Standard Library' (http://www.stlport.org/)

[Josuttis99] N. M. Josuttis, *The C++ Standard Library, A Tutorial and Reference*. Addison-Wesley, 1999.

[Raymond03] E. S. Raymond, *The Art of Unix Programming*, 2003. http://catb.org/~esr/writings/taoup/html/ch01s06.html#id2878339

[Robillard09] M. P. Robillard, 'What Makes APIs Hard to Learn? Answers from Developers', *IEEE Software*, vol. 26, no. 6, 2009. http://www.cs.mcgill.ca/~martin/papers/software2009a.pdf

[TDD] 'Test-driven development', accessed 2011-March-10. http://en.wikipedia.org/wiki/Test-driven_development

Why Computer Algebra Won't Cure Your Floating Point Blues

Numerical computing is proving quite a challenge. Richard Harris sees if a computer can do mathematics.

In the first article in this series we covered floating point arithmetic, its relatively benign rounding errors, its devastating cancellation errors and their slightly surprising order of execution sensitivity.

In the second article we moved on to fixed point arithmetic and found that it can suffer even more greatly than floating point arithmetic from these failure modes.

In the third article we covered rational numbers and found that when dealing with non-linear equations we must make a decision about how accurately we wish to approximate their results and consequently expose ourselves to exactly the problems we have with floating point numbers.

Computer algebra

So, again, can we do any better?

Well perhaps we could explicitly manipulate mathematical formulae rather than approximately evaluate them at each step of a calculation. For example, when taking the square root of 2 we should return a result that represents the operation itself rather than its result; something along the lines of "sqrt(2)". When the calculation is complete we could then evaluate it to any precision we desire. We will have effectively moved from arbitrary precision to infinite precision and will thereby have addressed all of the weaknesses of our other numerical representations.

Manipulating string representations of formulae would be rather unwieldy, so instead we shall represent them with trees. For example, the formula for the golden ratio

$$\frac{1}{2}(1+\sqrt{5})$$

can be represented by the tree given in Figure 1.

The nodes of the tree should be interpreted as the application of the operation they contain to the results of the nodes below it, with the leaf nodes being equated to the numbers they contain.

Such representations are often referred to as expression objects since the values they contain capture complete expressions rather than their results.

An expression class

We begin with a base class which will represent an abstract expression, as shown in Listing 1.

Naturally, we shall need a virtual destructor to ensure that derived objects are properly cleaned up.

The **approx** function shall compute the result of the expression using double precision floating point arithmetic whereas the **exact** method shall return the *n*th decimal digit of the result of the expression, with negative *n* being to the right of the decimal point and non-negative *n* to the left. We shall use the **empty** constant to indicate that there are no further digits to the left and, in the event that we can exactly represent a number, to the right of the decimal point. Note that we shall not allow leading or trailing zeros for any result other than zero itself, which shall have a zero digit at the zero'th position and shall have all other digits equal to **empty**.

Finally, the **sign** function shall indicate the sign of the expression.

The first thing we shall need is a derived class to represent the leaf node constant arbitrary precision integer values. Naturally, we shall use **bignum**s [Harris10] since we need to be capable of representing any integer, no matter how large. Listing 2 provides the definition of this class.

Note that since we shall treat this as an object type, and hence interact with it through pointers, we can afford to make the member data const and public. We shall deal with assignments by replacing the objects representing expressions rather than changing their states.

The constructor and destructor are straightforward, as shown in Listing 3.

The **approx** member function isn't so very much harder to implement, provided we don't care too much about recovering every last digit of

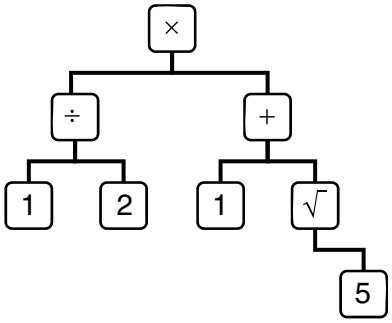


Figure 1

```
class expression_object
{
public:
  enum{empty=0xE};
  virtual ~expression_object() = 0;
  virtual double approx() const = 0;
  virtual unsigned char
     exact(const bignum &n) const = 0;
  virtual bignum::sign_type sign() const = 0;
};

expression_object::~expression_object()
{
}
```

Listing 1

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.

I'm reasonably happy to trade a relative error of the double precision epsilon in return for such a simple and efficient implementation

```
class integer_expression : public
expression_object
{
public:
  explicit
     integer_expression(const bignum &value);
  virtual ~integer_expression();

  virtual double approx() const;
  virtual unsigned char
     exact(const bignum &n) const;
  virtual bignum::sign_type sign() const;
  const bignum value;
};
```

Listing 2

```
integer_expression::integer_expression(
   const bignum &value)
: value(value)
{
}

integer_expression::~integer_expression()
{
}
```

Listing 3

precision and are a little blasé about overflow. A naïve implementation is given in Listing 4.

Here we simply iterate backwards over the digits of the **bignum** accumulating the sum of their values multiplied by the scale implied by their position.

Note that since the sum of the less significant digits might result in a carry, the true approximate result might require adding 1 to the least significant bit of the mantissa. That said, I'm reasonably happy to trade a relative error of the double precision epsilon in return for such a simple and efficient implementation.

If a call to **pow** overflows we shall have a result of plus or minus infinity, which isn't so very bad since infinity is a pretty good approximation of a number too big to fit into a **double**.

The **exact** member function is even simpler, albeit rather less efficient, provided we have an integer equivalent to the **pow** function for **bignum**s, as shown alongside the **sign** member function in Listing 5.

It would be extremely irritating to interact with this numeric type by directly managing object pointers. To avoid having to do so, we shall use a wrapper class, as shown in Listing 6.

Note that we shall keep track of our expression objects with a reference counted **shared_ptr** such as the one found in the Boost library.

```
double
integer_expression::approx() const
{
  typedef bignum::data_type::const_iterator
     const_iterator;
  double x = 0.0;
  double y = 1.0;
  size_t i = value.magnitude().size();

  const_iterator first =
     value.magnitude().begin();
  const_iterator  last =
     value.magnitude().end();

  while(first!=last && x!=y)
  {
    const double m =
      pow(double(bignum::mask)+1.0,
         double(--i));

    y  = x;
    x += m * double(*--last);
  }

  return
     (value.sign()==bignum::positive) ? x : -x;
}
```

Listing 4

The constructors are, as has consistently been the case, relatively straightforward as shown in Listing 7.

Note that, for convenience, we treat uninitialized or null initialised expressions as being equal to 0.

```
unsigned char
integer_expression::exact(const bignum &n) const
{
  if(n<0L) return empty;
  const bignum m = pow(bignum(10L), n);

  if(m>value) return n==0 ? 0 : empty;
  return (value/m).magnitude().front() % 10;
}

bignum::sign_type
integer_expression::sign() const
{
  return value.sign();
}
```

Listing 5

It would be extremely irritating to interact with this numeric type by directly managing object pointers

```
class expression
{
public:
  typedef shared_ptr<expression_object>
     object_type;
  enum{empty=expression_object::empty};
  expression();
  expression(const bignum &x);
  explicit expression(const object_type &x);
  double approx() const;
  unsigned char exact(const bignum &n) const;
  bignum::sign_type sign() const;
  object_type object() const;
  int compare(const expression &x) const;
  expression & negate();
  expression & operator+=(const expression &x);
  expression & operator-=(const expression &x);
  expression & operator*=(const expression &x);
  expression & operator/=(const expression &x);

private:
  object_type object_;
};
```

Listing 6

```
double
expression::approx() const
{
  assert(object_);
  return object_->approx();
}
unsigned char
expression::exact(const bignum &n) const
{
  assert(object_);
  return object_->exact(n);
}
bignum::sign
expression::sign() const
{
  assert(object_);
  return object_->sign();
}
expression::object_type
expression::object() const
{
  return object_;
}
```

Listing 8

The approximate and exact evaluation functions and the data access method are similarly simple and are given in Listing 8.

The `compare` member function can also be implemented quite simply, provided we are comfortable with the expense of subtracting two expressions during its calculation, as shown in Listing 9.

Now, this isn't going to work until we implement the arithmetic operators, so we shall get right to it!

Unfortunately, these are a little more complicated to get working properly for exact evaluation. We shall, therefore, implement just approximate evaluation for now as an indication of the general approach and we shall return to exact evaluation later.

Approximate evaluation

We shall use subtraction as an example since we're already using it; the remaining operators will be more or less the same.

The class definition for the subtraction expression is provided in Listing 10.

As ever, we have a trivial constructor and destructor, given in Listing 11.

```
expression::expression()
 : object_(new integer_expression(0L))
{
}
expression::expression(const bignum &x)
 : object_(new integer_expression(x))
{
}
expression::expression(const object_type &x)
 : object_(x ? x :
    object_type(new integer_expression(0L))
{
}
```

Listing 7

```
int
expression::compare(const expression &x) const
{
  const expression d = *this - x;

  if(d.exact(bignum( 0L))==0 &&
     d.exact(bignum( 1L))==empty &&
     d.exact(bignum(-1L))==empty)
  {
    return 0;
  }
  return d.sign()==bignum::positive ? 1 : -1;
}
```

Listing 9

it is but a short step to implement an expression object to represent algebraic variables

```
class subtraction_expression :
   public expression_object
{
public:
  subtraction_expression(const expression &lhs,
                         const expression &rhs);
  virtual ~subtraction_expression();
  virtual double approx() const;
  virtual unsigned char
     exact(const bignum &n) const;
  virtual bignum::sign_type sign() const;

  const expression lhs;
  const expression rhs;
};
```

Listing 10

```
subtraction_expression::subtraction_expression(
   const expression &lhs, const expression &rhs)
   : lhs(lhs), rhs(rhs)
{
}

subtraction_expression::~subtraction_expression()
{
}
```

Listing 11

```
double
subtraction_expression::approx() const
{
  return lhs.approx() - rhs.approx();
}

unsigned char
subtraction_expression::exact(
   const bignum &n) const
{
  throw std::runtime_error("");
  return empty;
}

bignum::sign_type
subtraction_expression::sign() const
{
  throw std::runtime_error("");
  return bignum::positive;
}
```

Listing 12

```
expression &
expression::operator-=(const expression &x)
{
  object_ = object_type(
     new subtraction_expression(*this, x));
  return *this;
}
```

Listing 13

The `approx` function simply approximately evaluates `lhs` and `rhs` and subtracts the `double` resulting from the latter from that resulting from the former, as shown in Listing 12 together with the exact evaluation methods that shall, for now, throw an exception.

Note that the return statements aren't strictly necessary, but they keep my compiler happy.

We use this class in the implementation of the subtraction operation of the expression class, as given in Listing 13.

Note that we shall effectively use the C++ operator precedence rules to implicitly build the expression tree, saving us from the tedious task of building it explicitly.

For example

```
x+y*z
```

will be translated as

```
operator+(x, operator*(y, z))
```

which, assuming we implement the free arithmetic operations in terms of the in-place arithmetic operations, would result in

```
expression(x)+=(expression(y)*=z)
```

and hence the required expression tree.

Rearranging expressions

An immediate advantage of such an approach is that by examining the run-time type information of the underlying expression objects we can transform one expression tree into another, simpler one that has an identical value.

For example, we could implement a `simplify` function that could manipulate the terms in an expression looking for a simpler representation. Using such a function we might expect

```
assert(simplify(x*y/x).object()==y.object());
```

to pass for `expression`s `x` and `y`.

Expression variables

From here it is but a short step to implement an expression object to represent algebraic variables and I should like to explore the ramifications of this before discussing the exact evaluation of expressions. Listing 14 illustrates just such a class.

computer algebra systems are immune to the problems that arise from cancellation error when approximating it using finite differences

```
class variable_expression :
   public expression_object
{
public:
  explicit variable_expression(
     const expression &value);
  virtual ~variable_expression();

  virtual double approx() const;
  virtual unsigned char
     exact(const bignum &n) const;
  virtual bignum::sign_type sign() const;

  const expression &value;
};
```

Listing 14

```
variable_expression::variable_expression(
   const expression &value) : value(value)
{
}

variable_expression::~variable_expression()
{
}
```

Listing 15

```
double
variable_expression::approx() const
{
  return value.approx();
}

unsigned char
variable_expression::exact(const bignum &n)
const
{
  return value.exact(n);
}

bignum::sign_type
variable_expression::sign() const
{
  return value.sign();
}
```

Listing 16

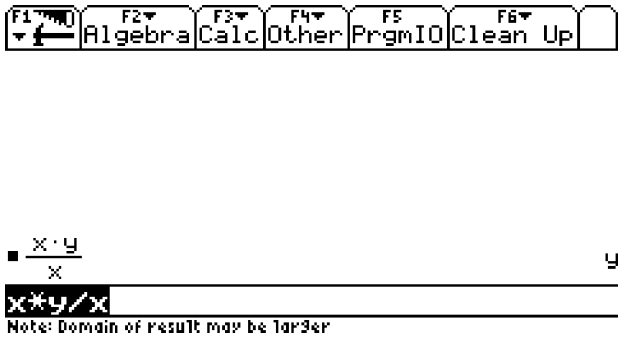


Figure 2

The constructor and destructor are given in Listing 15. Note that we can give the variable a value by assigning to the **`expression`** that it holds a reference to.

The evaluation member functions simply forward to the **`value`** reference, as shown in Listing 16.

Computer algebra systems

Now that we have an algebraic variable, we can represent algebraic expression and apply **`simplify`** to them too. In this sense expression objects lie at the heart of computer algebra systems such as Mathematica, MathCAD and even some top of the range calculators.

Figure 2 illustrates the result of simplifying $x \times y \div x$ on my own calculator [Texas].

We can go further still; if **`simplify`** can transform an expression tree than why not **`differentiate`**, or **`integrate`** or any other algebraic manipulation for that matter?

Figure 3 illustrates the calculation of the derivative of e^x at 1 and conclusively demonstrates that computer algebra systems are immune to the problems that arise from cancellation error when approximating it using finite differences.

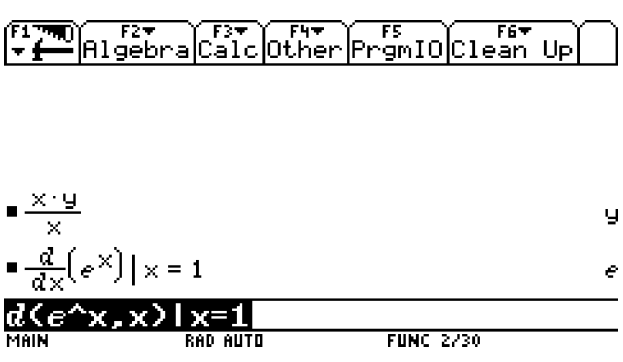


Figure 3

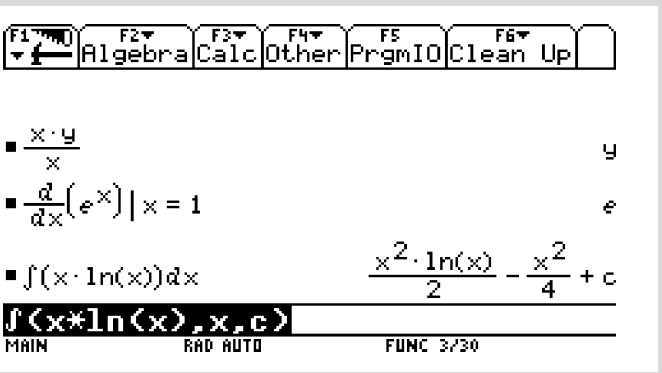


Figure 4

Figure 4 illustrates the calculation of the indefinite integral of $x \ln(x)$. This is quite a tricky integral unless you are familiar with the technique of integration by parts, which my calculator evidently is.

To check that this is the correct answer we need only differentiate it and confirm that we get the expression being integrated.

$$\begin{aligned}\frac{d}{dx}\left(\frac{x^2\ln x}{2}-\frac{x^2}{4}+c\right) &= \frac{d}{dx}\left(\frac{x^2\ln x}{2}\right)-\frac{d}{dx}\left(\frac{x^2}{4}\right)+\frac{d}{dx}(c)\\ &= \left(\frac{2x\ln x}{2}+\frac{x^2}{2x}\right)-\left(\frac{2x}{4}\right)+(0)\\ &= x\ln x+\frac{x}{2}-\frac{x}{2}\\ &= x\ln x\end{aligned}$$

Unfortunately, however, closed form results may be impossible to achieve in general. As an example, consider the expression

$$\int_{-\infty}^{c} e^{-x^2}\,dx$$

My calculator's evaluation of this reflects the fact that it has no closed form solution, as shown in Figure 5.

In practice computer algebra systems are extremely good at applying lengthy sequences of relatively simple manipulations but tend to struggle when more subtle sequences of transformations are required.

Exact evaluation

Now let's finally return to the issue of exact evaluation.

I'm afraid that I must admit that I'm not entirely sure how to do it. Presumably we shall need to implement algorithms that can perform numerical operations to arbitrary precision and cache any working data so that we can extend the number of digits without recalculating those that we have already found.

This might not be too unreasonably difficult for basic arithmetic, but I suspect that implementing numerical algorithms such as integration and differentiation might prove a little trickier. By trickier, I naturally mean demonstrably impossible in general.

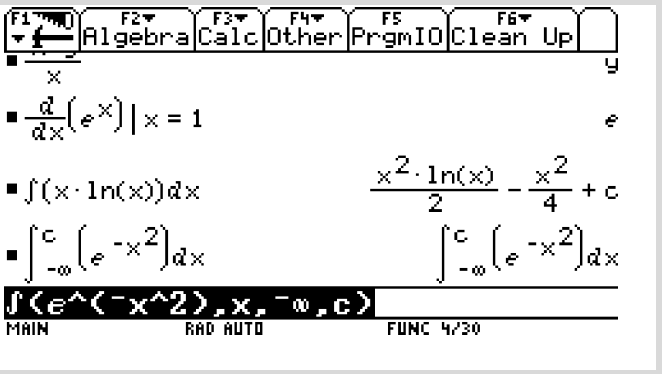


Figure 5

Unfortunately there is one major problem that such approaches cannot address. Recall that in order to make comparisons tractable we mandated that exactly representable numbers must have no trailing zeros. If they are not terminated with an **`empty`** value we cannot know that we have exhausted every non-zero digit after the decimal point.

For many expressions it is not a simple task to determine if a digit should have an **`empty`** value. As an example, consider the expression

$$\sin^2 x + \cos^2 x - 1$$

This is exactly equal to zero for every value of x, but the first two terms won't be exactly representable as a decimal fraction for almost all values of x. The addition of the first two terms shall inevitably be trapped in the endless production of zeros, hopelessly searching for a non-zero trailing digit.

The only way we shall escape this fate is to implement a full blown computer algebra system that can simplify all such difficult expressions into forms that can be computed in finite time.

No such system currently exists and, I am sorry to report, no such system ever will.

In 1931 Kurt Gödel proved that there are either infinitely many mathematical propositions that cannot be proven or disproven, or that it is possible to prove propositions that are false and that consequently the rules of mathematics are internally inconsistent [Gödel31].

This caused something of a stir in the mathematical community, who had hitherto been labouring under the illusion that both everything that was true could be proven and that everything that could be proven was true.

Modern mathematicians have come to terms with the fact that there are unprovable truths, or more accurately that there are undecidable propositions, mainly because the alternative is far too bitter a pill to swallow; internal consistency is simply too important to sacrifice.

Alan Turing took up the torch when he settled the decision problem and demonstrated that it wasn't always possible to determine whether a proposition was decidable or not [Turing37].

Expression objects are therefore superducks; capable of meeting all of our numerical requirements in a single bound but, like their counterpart Superman, are unfortunately wholly fictional in practice.

Quack, quack and away! ■

References and further reading

[Gödel31] Gödel, K., 'Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme', *I. Monatshefte für Mathematik und Physik*, vol. 38, pp. 173-198, 1931.

[Harris10] Harris, R., 'You're Going to Have to Think; Why Fixed Point Won't Cure Your Floating Point Blues', *Overload 100*, ACCU, 2010

[Texas] Texas Instruments Voyage 200

[Turing37] Turing, A., 'On Computable Numbers, with an Application to the Entscheidungsproblem', *Proceedings of the London Mathematical Society*, Series 2, Vol. 42, pp. 230-265, 1937.

The Agile Spectrum

Very few teams are truly Agile. Allan Kelly looks at the range of styles.

Agile is a broad church. It includes a lot of tools and techniques, some applicable to some teams and environments and others elsewhere. Anyone who thinks hard about how to measure Agility quickly realises it cannot be measured by adoption of practices, it needs to be considered on outputs and abilities.

Agile is sometimes simply defined as 'not waterfall'. This is a poor, if understandable, definition. Unfortunately, this means that any process that doesn't strictly follow the classic waterfall methodology can be considered Agile. Adding to the confusion 'Waterfall' can cover a number of different approaches, such as stage gate models like DoD 2167 and 2168 and all encompassing methods like SSADM.

In companies where strong, documentation centric, procedures have been hoisted on development teams, Agile is sometimes seen as a 'get out of jail free' card. Simply saying 'this project is Agile' is seen to exempt work from company procedures. Unfortunately, this card is also used as a cover for cowboy development.

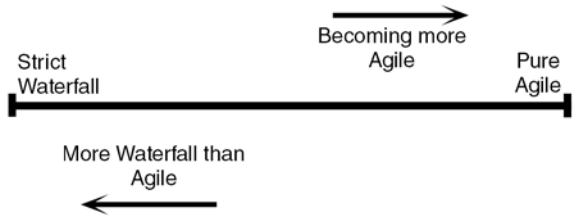
In truth there is a spectrum with strict-waterfall at one end and 'pure Agile' at the other – see Figure 1. Since waterfall never really worked that well, very few teams are at the strict waterfall extreme. In his analysis of software development projects over 20 years, Capers Jones suggests that, in general, requirements are only 75% complete when design starts, and design is a little over 50% complete when coding starts [Jones08]. He goes on to say that, as a rule of thumb, each stage overlaps by 25% with the next one.

It would seem reasonable that the pure Agile end of the spectrum is equally sparsely populated. Whether because few teams need to be so extremely Agile, or whether because experience and tools have yet to allow such a degree of Agility, some staged elements exist in many projects.

More than one software development team has encountered the situation when the team want to be more 'Agile', the organization and management might even be asking them to be more 'Agile', but there are still many 'requirements' in a big document and the expectation is that all these will be 'delivered'. Experience and anecdotal evidence suggest this scenario is faced by many teams.

This mismatch arises when the organization is largely waterfall but the development team are trying to work Agile. I have consulted with companies where senior managers believe Agile is only a delivery process for developers. Business case, requirements, design and even testing is waterfall, just the bit in the middle is Agile.

Allan Kelly has held just about every job in the software world. Today he provides training and coaching to teams and companies in the use of Agile and Lean techniques to develop better software with better processes. He is the author of *Changing Software Development: Learning to become Agile*, numerous journal articles and is currently working on a book of Business Strategy Patterns.
Contact him at http://www.allankelly.net.



The spectrum from Strict Waterfall to Pure Agile: almost everyone is somewhere inbetween

Figure 1

This article attempts to both understand the different degrees of Agility and provide teams with a way of resolving the requirements-delivery mismatch.

Three Agiles

On close inspection Agile has at least three styles: iterative, incremental and evolutionary, shown in Figure 2. These are largely governed by the development team's relationship with the requirements, and whether the organization wants work defined in advance or prefers goal directed working.

As we shall see in a moment, these three styles occupy different places on the spectrum. But, in truth, there is no clear cut divide between iterative and incremental, incremental and evolutionary, or even iterative and evolutionary. The three styles all overlap and fade into one another.

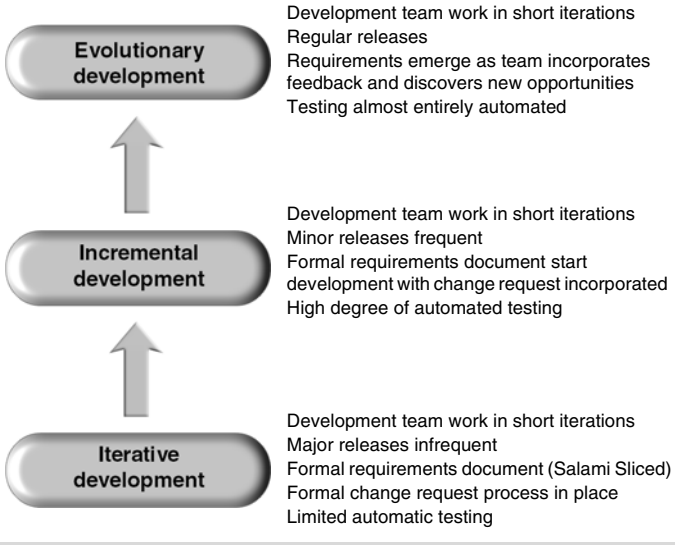


Figure 2

Some slices of salami will be thicker than others but that's just the nature of the world. Over time, with more skill at slicing salami it will improve and slices will be thinner

Iterative development – salami Agile

> Working in bite-sized chunks from predetermined requirements with one big delivery at the end.

Iterative Agile refers to the practice of undertaking projects in small, bite-sized chunks. Every two-weeks (or so) an iteration completes and the total amount of work is burnt down on a chart. Customers will probably be shown the latest version of the software at the end of the iteration, although this is little more than a demo. Most likely there will be a single software release at the end of the work – followed by several 'maintenance' releases.

At the start of work there is a big requirements document – the work to be done is, at least in theory, defined in advance. Someone, perhaps a previous project, perhaps external consultants, has created a list of the features and functionality the new system must, or should, have. The development team are expected to deliver all of it, or nothing.

The approach here is to see the big requirements document as an uncut sausage of Salami (long and dense). Someone on the team – preferably someone with Business Analysis skills but it could be a developer, project manager, or someone else – needs to slice the requirements into thin pieces of salami (story) for development.

There is no point in slicing the whole salami in one go. That would just turn a big requirements document into a big stack of development stories. The skill lies in determining which bits of the document are ready (ripe) for development, which bits are valuable, and which bits can be delivered independently.

Some slices of salami will be thicker than others but that's just the nature of the world. Over time, with more skill at slicing salami, it will improve and slices will be thinner.

Working in this fashion opens up the ability to accept change requests relatively easily. But because the work has been set up as a defined project with 'known' requirements these opportunities probably aren't exploited to the full. Similarly, opportunities to remove work will also appear – some slices of salami may be thrown away – but again this will depend on how rigidly the project seeks to stick to the defined work.

As well as the requirements document there are probably some estimates somewhere – maybe even a Gantt chart, which has to be updated to maintain the illusion that it is useful.

However, this is the land where the burn-down chart reigns supreme. There is a nominal amount of work to be done and with each iteration there is a little less. Such empirical measurement is likely to provide a good end-date forecast.

Salami Agile is the basis for incremental development and occurs somewhere about the middle of the spectrum. To go further towards pure Agile, work has to be based less on a shopping list of features and more on the overarching overall objective of the work.

Incremental development

> Working in bite-sized chunks from predetermined requirements with regular deliveries and accepting changes

Salami slicing is still prevalent in incremental development, at least during the early stages. Work is completed in bite-sized chunks and periodically delivered to customers to use. These events might, or might not, occur in tandem. While a team might work in two-week iterations, deliveries might only occur every two months.

The pieces of salami are delivered to the customer early, and over time customers start to realize they don't need some things in the original requirements document, so some slices can be thrown away and some salami left unsliced and unused.

This model capitalizes on the flexibility provided by eating salami rather than steak. Requirements which were not though of can be easily incorporated, others can be changed, enlarged or shrunk.

The iterative approach still assumes the original requirements are correct, so not implementing them all, or changing what is done, is a sign of earlier failure. In incremental development changes are seen positively and reductions in scope are seen as savings – a sign the model is working.

That real live users are getting access to the software early is valuable to the business. It also means user insights and requests are inevitable. Still, there is a major requirements definition somewhere, and while the team can accept change requests easily, it is still expected that one day the team will be done.

Burn-down charts might still be used to track progress, but at times they may appear as burn-up charts as work is discovered.

Tensions arise when the team are instructed to refuse changes, or themselves insist on continuing to salami slice the original requirements document, but users and customer are asking for changes based on their experience. In other words, the users and business have changed their understanding, but the team do not, or are not allowed to, change theirs.

There is no hard and fast line between iterative and incremental, they are just points on the spectrum – with incremental to the right of iterative by virtue of delivering more often. Perhaps the hallmark of incremental is that the team delivers on a regular schedule. When each delivery is a big deal, a special occasion, then things are really just iterative with occasional drops.

Evolutionary Agile – goal directed projects

> Working in bite-sized chunks from emerging requirements with regular deliveries

Evolutionary Agile takes this to the next level and is the natural home of goal directed projects. Teams start work with only a vague notion of the requirements. Over time the needs, practices and software evolve. As the software is released to customers the needs are reassessed, new requirements discovered, existing ones removed and new opportunities identified.

The team has a goal, will determine what needs doing (requirements) and do it (implementation) as part of the same project. The team is staffed with a full skill set to do the complete work – analysts, developers, testers and more. The team is judged and measured by progress towards the goal and

Many organizations, rightly or wrongly, consider any development process that is iterative in nature to be 'Agile'

value delivered, rather than some percentage of originally specified features completed.

Even goal directed Agile needs to start by establishing a few initial requirements. Some teams call this period 'sprint zero' in which a few seed stories are captured, from which product development (coding) can start as soon as possible. From there on, requirements analysis and discovery proceed in parallel with creation. Those charged with finding the requirements (Product Owners, Product Manager, Business Analysts or who-ever) work just a little ahead of the developers.

Burn-down, even burn-up, charts have little meaning for goal directed work because the amount of work to be done isn't known in advance. Work to-do and work done are better tracked with a cumulative flow diagram showing the progress in both discovering needs and meeting needs.

Governing goal directed work is superficially more difficult because it is not measured against some nominal total. Instead work needs to be measured against progress towards the goal.

These projects should be placed under a portfolio management regime that regularly – at least quarterly – reviews the progress and value delivered so far against the goal and the costs incurred. These figures should be produced within the team itself, and the team should feel confident enough to suggest its own end.

Taken together

Adding these points to the spectrum gives Figure 3. For a team migrating to Agile the objective is to move from left to right. These three approaches might reflect three levels of capability but they may also reflect the nature of Agile in a particular organization. One size does not fit all: some teams are better off with one style of Agile, and some with another.

Many organizations, rightly or wrongly, consider any development process that is iterative in nature to be 'Agile'. Therefore, in common parlance any method on the right of this spectrum is called Agile, while anything on the left is called Waterfall.

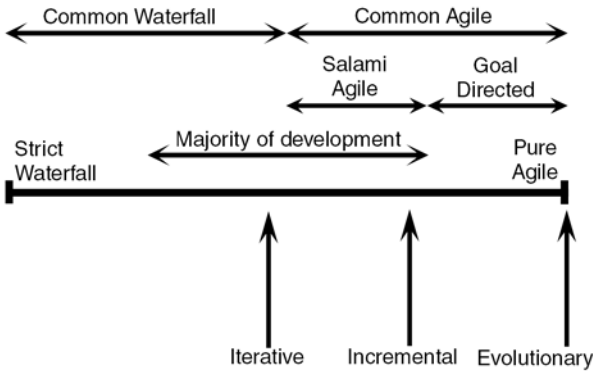


Figure 3

Waterfall approaches might split work into stages, work packages, or sub-projects which can make work look a little like iterative development. Although Waterfall development is associated with Big Bang releases many such projects are released in several Small Bangs. And after release 'maintenance' teams would continue to release updates.

Just as few teams actually embrace 100% evolutionary development, few teams ever followed a pure Waterfall approach. Indeed, I would argue that the Waterfall is so fundamentally flawed that a pure Waterfall was always impossible. (Before writing to take me to task please read the original Waterfall paper [Royce70].)

In my experience most development projects lie somewhere between these two extremes, mostly clustered around the centre. Although I don't have any data to support my argument I suspect that a standard-distribution bell-curve could be laid over this diagram, reflecting that most teams following an interactive process, with a few teams more incremental and a similar number doing periodic releases on a Waterfall basis.

While there are no hard and fast rules about when a team is doing one style of development or another, there are some common traits visible by looking at the practices the teams adopt. These are summarised in Table 1. While these attributes are a useful way of describing and comparing different styles and different teams, they are not prescriptive.

Examples

Interestingly, there is one area of software development where the goal-directed evolutionary approach has long been the norm: maintenance. Maintenance teams have the goal of keeping systems working, fixing bugs and, often, small enhancements. Work emerges over time and the highest priority work gets done and other work is left undone.

I remember working on a financial reporting tool called FIRE in 1997. There was no roadmap or even a plan for the product. The company had three, four, then five and even six customers. As each sale was made, new requirements emerged: port from Solaris to Windows, from Sybase to SQL Server, to Oracle, to AIX. And of course bugs.

These requests arrived with more or less noise and urgency. I introduced time-boxed iterations to the team: we released each month, and put a white board on the wall to show what we were doing. Each iteration had a collection of work: we delivered, and then reviewed what had arrived in the last month.

Evolutionary would be the best characterisation of FIRE. Requirements and processes emerged as the work progressed. The overall goal was never clearly stated and we only had elementary unit testing – but we had some!

Conversely, one of my clients in Cornwall is currently writing a completely new version of their flagship product in an iterative way. The feature list is almost entirely taken from the existing product. The team work in one-week iterations. At the end of each iteration their proxy-customer reviews the work and ticks it as done.

The work to do is grouped – physically – into monthly bundles – November, December, January, February. The original aim was of

releasing in March but it now looks like it will be April. Nothing will be released until it is all done.

Of course once the first release is done working will change. Probably the team will take more of an incremental approach with monthly updates. They still have plenty of features – new or held over – to continue implementing for a few months. I expect that at some stage new requests and ideas will bring a more evolutionary nature to the work.

This team will need to revisit their overarching goal. As I write the goal is 'Get a version released with a subset of the current features'. At some time in the near future they will need to question that goal lest they drift into a 'find work, do work' mentality.

A change model

It is useful to consider this spectrum as a change model. Assume a starting point somewhere on the left of the spectrum, a team doing some form of common waterfall with all the imperfections that suggests. Being Agile, by any definition means moving to the right.

As a first step the team can adopt an interative approach and use Salami Agile to manage requirements. In time, as they improve, they advance to an incremental approach. To go further the team needs to move away from salami and become goal directed. This requires more of the organization to embrace the Agile ways of the team. Some teams may stall here for this reason.

When a team has a proven track record at incremental delivery, the organization will come to trust the team, then opportunities arise for goal directed, evolutionary work.

Summary

Although Waterfall and Agile are often characterised as straight alternatives, neither is particularly well defined. It is better to view them as representing different areas on a continual spectrum from a strict phased approached to a no-phased approach.

On the Agile end of the spectrum there are different ways of approaching work. Many teams work with pre-determined requirements in a salami fashion. They deliver software iteratively or incrementally. A few teams work in a more goal-directed fashion where needs, solutions and processes are evolving.

Different techniques, tools, practices and processes are used at different parts of the spectrum, but there are no hard and fast rules as to what is used when. ■

Acknowledgements

Thanks to Paul Grenyer and Ed Sykes for reviewing an early draft of this article; and the Overload editorial team for their usual attention to duty.

References

Jones, C. 2008. *Applied Software Measurement*, McGraw Hill.

Royce, W. W. 1970. *Managing the development of large software systems: concepts and techniques*.

Practices	Waterfall	Iterative	Incremental	Evolutionary
Stand-up meetings	No	Yes	Yes	Yes
Planning	Start of project; revisions as needed	Regular 2–4 week iterations	Regular 2–4 week iterations	Regular 2–4 week iterations
Status reporting	Regular, against plan	Regular	Regular	Regular, against goal
Retrospectives	Sometimes at end of work	Occasional – more talked about than done	Regular	Integral
Demo 'Show and Tell'	Occasional snapshot	Occasional	Regular	Only as information prior to release
Planning				
Budget	Allocated at start	Allocated at start	Mostly upfront	Arrives in increments
Budget control	Monitored against plan	Monitored against plan		Value delivered v. cost incurred monitored
Technical practices				
Releases	Once: at end	Once at the end, or at irregular intervals	Regular during project	Like clockwork
Automated unit testing	No	Maybe	Yes	Yes
Automated acceptance tests	No	No	Yes	Yes
Test first development (TDD)	No	Some	Lots	Everywhere
System integration tests	At end of project	During project	During project	Ongoing during project
User acceptance testing	Only end of project	At end of project	During project	Ongoing during project
Continuous integration	No	Yes	Yes	Yes
Tracking charts	Gantt	Burn-down	Burn-up	Cumulative flow
Design	Big up front activity	Mostly upfront	Some up front plus refactoring	Little upfront; mostly emergent with refactoring
Goal	Requirements are goal	Requirements are goal	Mix of upfront requirements and goal directed	Governs project and directs progress
Requirements	Officially specified in advance	Specified in advance; salami sliced to developers	Specified in advance; salami sliced to developers	Emerge during project
User feedback	Minimum	Little	Plenty but little scope to change incorporate	Fundamental to project success
Change control	Traditional – changes seen as problems	Traditional	Relaxed traditional	None – changes are requests

Table 1

On CMM, Formalism and Creativity

No Bugs requires us to improve software quality. Sergey Ignatchenko considers some of the potential problems.

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 us from providing an exact translation. In addition, both the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

Thlayli lay meth methrah nao
(Bigwig is a poor storyteller)

Creativity and formalism

Today I will try to touch on quite a sensitive issue, related to the subtle relationship between creativity and formalism, which my fellow rabbits can often feel but which is usually quite difficult to write down. While a few years ago it was argued (see, for example, [Konrad05]) that agile development can co-exist with formal methodologies like CMM, the question about the co-existence of formalism with creativity has not been addressed in the rabbit literature yet.

I will certainly not argue that creativity is always a good thing. In many cases I am personally really afraid of excessive creativity. For example, I certainly don't want to fly on a plane which has been serviced by mechanics who were excessively creative (such as the one described in [JA8119]), or to be operated on by a surgeon who's just had a fancy idea about how to make things so much better and wants to try it on me. On the other hand, in many cases creativity is highly desirable – the guy who invented the wheel did need to be creative.

Now to the question of formalism. Let's consider a team or organization which does need to be creative. What level of formalism is optimal for such a team? Should this team be in a state of complete anarchy? Or should it be a perfectly working machine where everything is done 'by the book'? Intuitively, it is quite clear that the latter is not the right way to inspire creativity, but unfortunately way too often management doesn't realize this and measures success, not in terms of a successful end product, but in terms of 'how organized the process is'. Let us consider this whole problem using CMM – Capability Maturity Model – as an example of such a management approach.

On CMM

CMM originated at the end of 1980s in a book by Watts Humphrey [Humphrey89]. From the very beginning, CMM was closely related to the US Department of Defense (and in particular the US Air Force). While the idea of improving software quality is certainly commendable, especially within military applications, it seems that the temptation to make developers march in formation was too strong to overcome, and this temptation eventually found its place within CMM. Later, around the end of the 1990s, CMM has been replaced with CMMI – Capability Maturity Model Integration – which tends to cover much more than the original, including 'CMMI for Development', a direct successor of the original CMM. Also the formally separate but ideologically similar 'People CMM' was released. Ironically, about the same time the very same Watts Humphrey realized that CMM doesn't really work in practice and came up with an alternative model known as 'Personal Software Process', which has evolved into 'Team Software Process', a.k.a. PSP/TSP. While the main promoter and owner of the 'CMM' trademark (SEI of Carnegie Mellon University) considers PSP/TSP as a valid (and recommended) CMM/CMMI implementation, for the purposes of this article we will consider it separate and specifically comment on it later.

From the point of view of management, CMM if often considered as a kind of 'holy grail', where it is enough to pre-build organizational processes and procedures and then the project will march ahead towards the bright dawn of success; unfortunately, experience shows it is certainly not guaranteed. In addition, CMM (similar to ISO9001) is often seen as a prerequisite to obtaining certain government contracts, as well as a way to get some kind of certification to show clients that the organization is a 'good one'. Research conducted by SEI shows improvements in productivity, at least in some cases. On the other hand, it often faces harsh criticism (again, similar to ISO9001) from both developers and CIOs, ranging from 'CIOs who look to CMM for guarantees won't find them' [Koch04] and 'In fact, the study found that Level 5 companies on average had higher defect rates than anyone else.' [Koch04] to 'At worst, the CMM is a whitewash that obscures the true dynamics of software engineering, suppresses alternative models.' [Bach94] Opinions of fellow rabbit developers are often even harsher, which is why I won't be able to quote them here. In short, it is quite a controversial subject.

Repeatability and replaceability

The key idea behind CMM is *repeatability*: as [P-CMM] says, 'A fundamental premise of the process maturity framework is that a practice cannot be improved if it cannot be repeated'. Even this statement is not really obvious, but let's see where it leads. For the purposes of this article, we will ignore many other implications of *repeatability*, concentrating on only one of them: *for the process to be repeatable, people need to be replaceable*. CMM states it in terms of 'exceptional individuals' [P-CMM, page 12]: '...in low-maturity organizations, their results depend largely on the skills of exceptional individuals...' (and in CMM speak, 'low-maturity' is pretty close to 'double-plus ungood').

Now let's try to put ourselves in the shoes of a manager: I am a manager, and have an exceptional individual on the team, great! But after reading a book on CMM, I'm starting to wonder: what happens if she leaves? All repeatability goes out of the window. Therefore to comply with CMM I (as a manager) need to eliminate all *dependencies* on irreplaceable

'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

whenever creativity is necessary, a manager should not carve processes in stone and then fit people into those processes

Exceptional individuals

It is important to note that even if there are many exceptional individuals on team, replaceability is still problematic if processes are rigid. *Exceptional individuals* are exceptional in *different ways*, so even when replacing one exceptional individual with another one is possible, it is not a direct replacement: as experience of fellow rabbits shows, in the vast majority of cases replacing an exceptional software developer (with another exceptional developer) inevitably leads to substantial changes in the project processes, or project architecture, or both.

developers. If done with caution, it is not a bad thing to eliminate dependencies, but unfortunately way too often management misreads it and eliminates *exceptional individuals* completely, using CMM as an excuse. Add that it is way too often aligned with the inclinations of a not-so-good manager who is not competent enough to manage the project, and we see the classical case of *injelititis* [Parkinson57]: where '...the head of the organization is second-rate, he will see to it that his immediate staff are all third-rate; and they will, in turn, see to it that their subordinates are fourth-rate.' Regardless of whether it was intention or not of CMM to achieve such a situation, it certainly helps managers to reach it.

Exceptional vs average

Even if the manager doesn't suffer from *injelititis* and doesn't eliminate everybody who's smarter than him, applying CMM is problematic in certain areas. If processes are strictly adhered to (the thing which CMM strongly advocates), what will these *exceptional individuals* do within a CMM-compliant organization? By the definition of CMM compliance, such individuals are required to obey the same processes as everybody else, and if these processes are rigid enough there is no room to apply their *exceptional abilities*; as a rule of thumb, *exceptional individuals* either leave such organizations, or find a way to bypass the processes (which essentially ruins CMM). In short: if management tries to fit an *exceptional individual* into an existing process (making the *exceptional individual* just another 'cog in the machine' without regard to her *exceptional abilities*), it doesn't work.

Therefore, by going ahead with the very spirit of CMM towards repeatability and replaceability, it will inevitably eliminate exceptional individuals from the team, which leads towards teams consisting only of about-average individuals who're fine with the role of 'cogs in the machine'. The approach of about-average individuals works great in those fields where creativity is not an issue (there is no argument that air mechanics should be easily and directly replaceable), but what about areas where creativity is necessary? What about a replaceable Einstein or Enrico Fermi?

Are creativity and CMM are incompatible

While it is obvious at an intuitive level that the approach of average individuals won't work where high levels of creativity are necessary, it is interesting to note that it is possible to provide a more formal demonstration of it. Let's take a look at one of the criteria used to measure innovation, namely the granting of patents. In most jurisdictions, for patents (which are undoubtedly one way to acknowledge innovation) there is a so-called 'average engineer' test: usually, a patent cannot be granted if it is obvious to 'a person of ordinary skill in the art' (which is essentially legalese for an 'average engineer'). It implies that inventions cannot be made if you have only 'average engineers' on board. As we saw above, *replaceability* in practice means 'average engineers' (ie not using exceptional abilities of *exceptional individuals*, which is about the same thing), and as CMM implies *replaceability*, it should not possible to generate any patents within truly CMM-compliant organizations. Q.E.D.

Obviously, this 'proof' is not a strict one, and in fact some of organizations which are formally CMM-level 5 do produce patents; however my fellow rabbits who went through CMM and ISO9001 audits feel that was only because CMM compliance (just like ISO9001 compliance) is merely a formality which has nothing to do with realities of software development in an organization; therefore, formal compliance has nothing to do with adhering to the spirit of CMM. Also of interest is that, as with any text which is vague enough, all kinds of interpretations are possible, so it is perfectly feasible to create formally correct interpretations of 'what CMM means', with fundamentally different conclusions (which essentially is what was done when 'Personal Software Process'/'Team Software Process' were designed). What is important is that the interpretation described above is certainly by far the most common one when it comes to real managers.

Process-centered vs people-centered

At this point a much more important question arises: does all this mean that projects which require creativity should be completely non-structured, informal, and in a state of anarchy? Not really. It just means that teams which need creativity should be managed not in a process-centered way (like CMM advocates), but in a people-centered way. In other words, whenever creativity is necessary, a manager should not carve processes in stone and then fit people into those processes (essentially making people 'cogs in the machine'), but the whole paradigm of management should be completely opposite: one will need to build a team of individuals, then build processes around this team, and adjust the processes when the needs of the team (or the team itself) change. We can compare these two approaches in software project management ('process-centered' and 'people-centered') to two fundamentally different approaches in software development itself: a 'process-centered' approach, with processes essentially carved in stone, is similar to the (in)famous 'waterfall' methodology, while a 'people-centered' one implies dynamic processes with multiple iterations, similar to 'agile' development methods.

Such 'people-centered' approaches are in fact nothing new: despite being admittedly more difficult for the manager, they have been used many times with tremendous success. [VirtuosoTeams] (BTW, I strongly recommended it for fellow rabbits who're interested in team leading or management) describes in detail several such teams, including the team behind the 'Manhattan project'. With a dozen Nobel laureates on board, and a task of epic creativity proportions, it certainly wasn't anywhere close

it is perfectly feasible to create different and formally correct interpretations of 'what exactly CMM means'

We DO need CMM certification, and we DO need Creativity, are we in Real Trouble?

If your company really needs to get CMM certification, certainly the best bet would be to go with 'Personal Software Process'/'Team Software Process' (a.k.a. PSP/TSP). While they're not exactly 'people-centered', they're not exactly 'process-centered' either, so it is possible to keep a certain level of creativity within PSP/TSP. In addition, PSP/TSP (unlike agile methodologies similar to Scrum/XP) are recognized by CMM appraisers, so it might work as a reasonable compromise which allows both to reach required certification and to keep creative individuals.

to CMM past level 2. Still, Colonel Leslie Groves, while certainly struggling with the management of such a 'virtuoso team', had managed to find a way to make things work, and (despite his military background) his approach was certainly 'people-centric', and not 'process-centric' as prescribed by CMM. While the moral grounds of the 'Manhattan project' are debatable, from a technical and management point of view it certainly was a tremendous success. ■

References

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

[Bach94] 'The Immaturity of CMM', James Bach, *American Programmer*, 1994, http://www.satisfice.com/articles/cmm.shtml

[Humphrey89] Watts S. Humphrey *Managing the Software Process*, Addison-Wesley, 1989

[JA8119] http://en.wikipedia.org/wiki/Japan_Airlines_Flight_123

[Konrad05] 'Agile CMMI: No Oxymoron', Mike Konrad, James W. Over, 2005, http://www.drdobbs.com/184415287

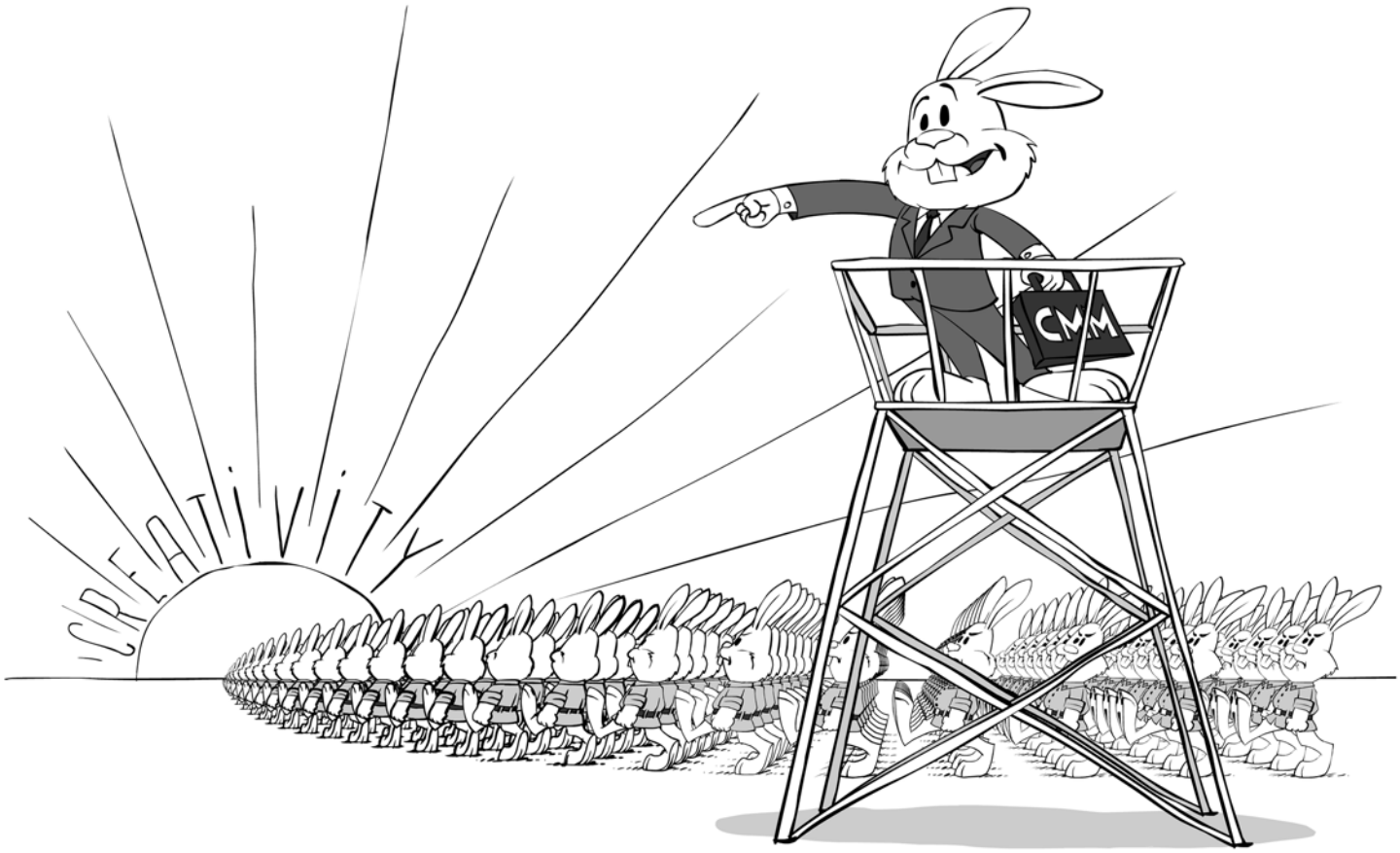
[Koch04] 'Software Quality: Bursting the CMM Hype', Christopher Koch, http://www.cio.com/article/32138/Software_Quality_Bursting_the_CMM_Hype?page=1&taxonomyId=3000

[Loganberry] David 'Loganberry', *Frithaes! – an Introduction to Colloquial Lapine!*, http://www.loganberry.furtopia.org/bnb/lapine/unit14.html

[P-CMM] *People Capability Maturity Model Version 2.0*, Second Edition

[Parkinson57] *Parkinson's law, and other studies in administration*, Houghton Mifflin, 1957. Chapter 8, Injelititis, or Palsied Paralysis

[VirtuosoTeams] *Virtuoso Teams: Lessons from teams that changed their worlds*, Andy Boynton, Bill Fischer, FT Press, 2005



Refactoring and Software Complexity Variability

Most code bases could have their complexity improved. Alex Yakyma presents a model that suggests how to do this.

The inherent complexity of software design is one of the key bottlenecks affecting speed of development. The time required to implement a new feature, fix defects, or improve system qualities like performance or scalability dramatically depends on how complex the system design is. In this paper we will build a probabilistic model for design complexity and analyze its fundamental properties. In particular we will show the asymmetry of design complexity which implies its high variability. We will explain why this variability is important, and why it can, and must, be efficiently exploited by refactoring techniques to considerably reduce design complexity.

Introduction

There are different views on refactoring in the software industry. Because refactoring is relatively neutral in respect to the choice of software development methodology, teams that practise Scrum or even Waterfall may apply refactoring techniques to their code. There are opinions on refactoring as a form of *necessary waste* (some authors elaborate on the concept of pure waste vs. necessary waste [Elsammadisy] attributing refactoring to necessary waste). This analogy often becomes ironic since many executives and software managers think that refactoring is in fact a pure waste and thus should not be undertaken by teams. At the same time there is the very valid standpoint regarding refactoring as a method of reducing the *technical debt* [Cunningham]. Some consider refactoring as a way of *entropy reduction* [Hohmann08]. The importance of this team skill on the corporate scale is explained in [Leffingwell10].

In all cases it is obvious that refactoring has to deal with something we call *software design complexity* – an overall measure of how difficult it is to comprehend and work with a given software system (add new functionality, maintain, fix defects etc.).

Let us start by analyzing complexity more deeply in order to understand how to cope with it.

The hypothesis of multiplicativity

We base our model of software design complexity on its multiplicative nature. Let's consider a list of factors that influence the complexity. It is not at all a full list and not necessarily in order of importance (applicable to OOP-based technology stack):

1. Meaningful, clear names of classes, methods, local variables etc.
2. Clarity of flow
3. Usage of 'typed' collections (1)
4. Usage of interfaces
5. Use of framework capabilities vs own implementations
6. Following single responsibility principle
7. ...

Obviously, some of the factors above are combinations of more specific independent factors. For example, factor #1 is a combination of naming clarity of methods, fields, classes, interfaces, packages, local variables. Factor #2 is also a list of specific factors: whether complicated conditional statements are reasonably decomposed, whether duplicate conditional fragments are reasonably consolidated, whether unnecessary control flags removed etc. It is easy to see that there is fairly large amount of such specific independent factors that determine the complexity.

Let's see what happens if we have a combination of any two factors. E. g. if we do not follow the single responsibility principle [WikiSRP] the code is harder to understand, debug, or maintain because objects of the same class can play considerably divergent roles in different contexts. At the same time not giving meaningful names to classes and their members makes code very hard to comprehend. A combination of these two has a multiplicative effect. To articulate this better let's use an example.

Assume we have class A with a vague class name and member names. Then the person that debugs the code and encounters objects of this class will have to cope with some complexity C of the class caused by naming problem of this class and its members (for simplicity sake think of C as the effort required to understand what A means in the context of our debugging episode). Let's now also assume that A fulfills 3 different responsibilities depending on the context. Then in order to understand the behaviour of A in this specific context of debugging you need to analyze what each of the names (class, field or method) would mean in each of the three possible contexts spawned by roles for class A. In other words, the complexity is $C \times 3$.

So we may consider overall complexity C as a product of a large number of individual factors:[1]

$$C = f_1 \bullet f_2 \bullet \ldots = \prod_i f_i \qquad (2)$$

The important characteristic of the factors above is that they are all *independent*. Taking into consideration the random nature[2] of these factors and assuming their fairly typical properties (more details below) we conclude that:

1. It is important to note that our model describes random behaviour of complexity at any arbitrary (*but fixed*) moment in time. In other words our model answers the following type of questions: what could be the design complexity of a product after the team works on it for, say, 2 months.
2. When we say that the complexity (or one of its components) is random, we mean that if we work on project X within a certain timeframe, the resulting codebase (as an 'evolving system') may end up in any one of a number of different possible states, each with a different level of design complexity.

Alex Yakyma provides Agile training and coaching to teams in North America, Europe and Asia, helping them to establish efficient development process and engineering practices. Contact him at www.yakyma.com

Assertion 1: Software design complexity is approximately[3] a lognormaly distributed random variable.

The sidebar contains the proof and is *optional* for a reader who wants to skip to the conclusions of this assertion.

The analytical expression for the probability density function (PDF)[4] of a lognormally distributed random variable can be found in [Lognorm] and is not of our current interest. Instead we will be more interested in its generic behaviour.

Finally, the only reason why we needed as set of *independent* factors in our factorization was to apply the Central Limit Theorem and thus prove that the complexity is a lognormal random variable. In their daily life teams deal with factors which influence each other. That way a team may have a good chance of controlling complexity with a reasonably small effort. The next section includes some examples.

Analyzing the model

For a given moment of time t_1 the graph of design complexity PDF looks like Figure 1.[5]

The *mode* is the most common observation, in other words it will be the most common outcome of a single given development project. The mean would be 'average' value if you repeat the 'experiment' multiple times (e.g. N teams work on N independent but identical projects). As follows from Figure 1, a lognormal distribution is 'skewed'. Unlike a normal distribution where *mean* = *mode* and the PDF would be symmetrical about the mean value (the well-known bell curve), in case of a lognormal random variable, 'smaller' complexities are 'compressed' on the left of the mode value while 'higher' complexities are scattered wide to the right. Actually we have a 'long tail' on the right side of the plot. These considerations imply that the result of a single observation will most likely be misleading. In other words:

Asymmetry of design complexity You are more likely to have a design complexity that is higher than the mode, and less likely to have one that is less than the mode. Occasionally design complexity will have extremely high values.

This fact sounds like a pretty sinister beginning of our journey, but the following two points mitigate the impact:

1. High variability of design complexity basically affects those teams that do not purposefully reduce complexity, and…
2. There is a reliable method of reducing complexity.

The method used is *refactoring*. It is easy to see that using our factorization above or similar, it is obvious what needs to be refactored to counter the effect of multiplicativity of design complexity and thus keeping complexity under control.

Note that in the factorization (1) we required that factors were independent. Although it was absolutely necessary for analysis purposes and proving that the complexity follows lognormal distribution, it is not at all required for your own strategy of refactoring. We may securely use 'overlapping' refactoring approaches if the team finds that convenient. The example of such dependent factors (and respectively the refactoring techniques) can be: 1) *Complex flag-based conditions in loops* – the factor, 'Remove Control Flag' – the refactoring method (see [Fowler99], p.245) and 2) *Unnecessary nested conditional blocks* – the factor, 'Replace Nested Conditional with Guard Clauses' as a refactoring approach (ib., p.250).

3. 'Approximately' means that if for a second we assume that there is not just 'large' but an infinite number of factors in (1) then expression (2) can be 'reduced' to C_n – a product of the first n factors. Our assertion basically states that the distribution of C_n tends towards lognormal as $n \to \infty$.
4. PDF – Probability Density Function of a continuous random variable is a function that describes the relative likelihood for this random variable to occur at a given point.

Proof of Assertion 1

Let's make two assumptions:

1) Random variables $\{\ln(f_1),\ln(f_2),\ldots\}$ have finite means and variances. This assumption makes sense from a practical standpoint (e. g. usage of typed vs untyped collections in the source code may vary but have an average ratio (of say 40%) with finite standard deviation (let's say, 20%); another example: method bodies would have an average length with deviation from the average approximately limited by its standard deviation etc.). Due to the fact that these factors have finite and relatively small range of values, we may conclude that their logarithms also have finite means and deviations.
2) Lindeberg's condition [L-Cond], which looks scary but actually means the fairly simple fact that the 'outliers' (that sit outside the 'circle' with radius composed of all variables' standard deviations) represent a minor set.

With this all said, we may apply Central Limit Theorem (in its version by Lindeberg and Feller [CLT-L-F]) to the sequence of random variables $\ln(f_1),\ln(f_2),\ldots$.

This gives us:

$$\frac{\sum_{i=1}^{n}\ln(f_i)-\sum_{i=1}^{n}\ln(\mu_i)}{\sqrt{\sum_{i=1}^{n}\ln^2(\sigma_i)}} \xrightarrow{d} N(0,1) \qquad (3)$$

meaning that the expression on the left of (3) converges to a normally distributed random variable (rv) *in distribution* (see [CNVRG] for more detail).

Here $\ln(\mu_i)=E[\ln(f_i)]$, and $\ln^2(\sigma_i)=E[(\ln(f_i)-\ln(\mu_i))^2]$ – mean and variance respectively.

But this means that the expression on the left in (3) is extremely close to normal distribution for big n. Let's fix some large integer n. Then remembering that for any positive real numbers a and b $\ln(a)+\ln(b)=\ln(ab)$ we have:

$$\ln\left(\prod_{i=1}^{n} f_i\right) \approx^{d} \alpha N(0,1)+\beta \qquad (4)$$

where α and β are constants (their meaning can be easily derived from (3)) and thus on the right side of (4) we have also a normally distributed variable ($\approx^d$ means that distribution functions of rv's are approximately equal, not the rv's themselves). This by definition means that C is approximately lognormal for big n.

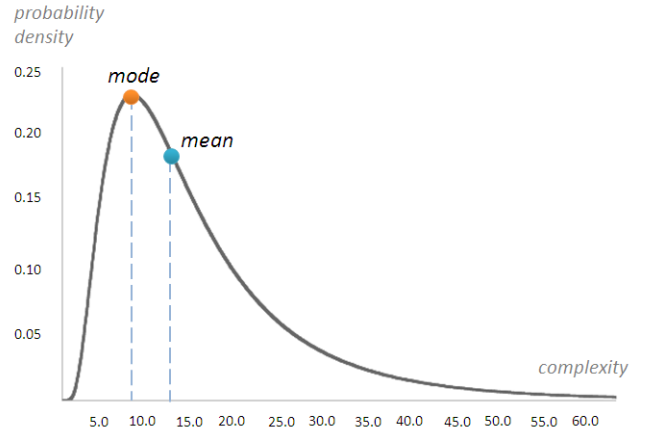


Figure 1

5. To better understand how the PDF changes with time: if $t_1<t_2$ then obviously the complexity at the moment t_2 is also lognormal but its graph is more 'stretched' lengthwise along the horizontal axis so that in particular its 'peak' is further to the right. This is more exact statement that the complexity tends to grow over time as development progresses. Though note that the dynamic analysis of design complexity is beyond the scope of this paper.

Obviously when you reasonably replace nested conditionals with 'guards', it will also affect some flag-based loops replacing their complex conditions with return statements where it is appropriate. So 1) and 2) affect each other to a certain extent but it is ok to use both as part of your strategy.

Another example shows that a team may choose additional factor that is not even on the list but which may influence other factors. This, for example, could be 'keeping methods reasonably compact'. In order to achieve this, the team will optimize flow control structures, data structures, reasonably use inheritance, single responsibility principle, use of framework/lib capabilities instead of own implementations and so on.

In fact refactoring is no less important than the creation of code in the first place. As Martin Fowler points out [Fowler99, p. 56-57]: 'Programming is in many ways a conversation with a computer… When I'm studying code I find refactoring leads me to higher levels of understanding that I would otherwise miss.'

Note that while we are aiming at reducing the complexity we still accept the fact that there is no way to avoid the variability of 'higher values' for it is an objective statistical law for this type of rv. In other words there is no way the team could turn a lognormal distribution into a symmetrical one, even though they are the best of developers.

Another important consequence of the design asymmetry for the economy of software engineering is that (because $mean \neq mode$) ***in the long run there is considerable hidden extra effort in maintaining the product***. Indeed the most probable outcome for complexity after one episode of development is by definition equal to the *mode*. But after being repeated multiple times it gravitates to the *mean* and we remember $mean > mode$ in the case of lognormal rv. Thus N such episodes yield the additional (and much worse – hidden) maintenance cost proportional to $N \times (mean - mode)$. This hidden extra effort can never be totally eliminated but can be reduced.

A team that purposefully refactors, either partially or totally, reduces the impact of certain individual complexity factors. The high variability means that you refactoring can dramatically succeed in reducing design complexity.

Refactoring means changes in design. These changes (sometimes dramatically) modify the information flows within the system, re-organizing and re-distributing information in different ways which leads to uncertainty and introduces variability to the outcome. Reinertsen [Reinertsen09] points out the exceptional importance of variability in the economics of product development. In our case the outcome of refactoring is also quantifiable – it is a team's velocity in delivering user value. While refactoring utilizes the variability, unit testing keeps refactoring within the limits. Unit tests bring considerable certainty to the scene: when you change few lines of code and then make sure your tests still run – this means that system functionality is not or almost not broken at all and changes in design did not lead to a wrong design.

> Unit testing and refactoring used in conjunction sustain the balance of variability and help utilize this variability for implementing effective design.

Even though unit testing is an engineering practice that represents huge independent value, it has many important nuances in context of system refactoring applied to reducing the complexity:

- Unit testing should be a continuous effort and go hand-in-hand with refactoring. Changes in code often infer change of method interfaces and logic. So when tests don't run it may mean two things: 1) that the logic/interface is wrong or 2) code has changed and requires modification to the unit tests as well.
- Unit testing does not constrain all areas of system refactoring. It is impossible to unit-test such things as clarity of naming or whether or not open/close principle is being followed. But even when renaming a method or local variable, tests guard us from breaking the logic.
- Unit testing allows for refactoring at any point in time. We can return to a specific fragment of code after a while and safely refactor it.
- Unit testing and refactoring mutually enable each other. It is very hard to unit-test a jsp page that performs direct calls to a database, handles business logic and prepares results for output. Instead, separation of concerns allows for better testability.

It is important to know that because of the high variability of complexity and the ability of refactoring to dramatically reduce one, refactoring becomes an extremely important *competitive advantage* of software teams.

Summary

Software design is usually more complex than we may think and factors like long methods or ambiguous names are just a few examples of a long list of forces that dramatically increase the complexity. Although the asymmetry of design complexity means that high complexity is more probable, it also gives teams a clue of how to exploit this asymmetry to reduce it. Continuous purposeful refactoring reduces the complexity at the same dramatic 'rate' and is necessary to sustain software maintainability in the long haul. ■

References

[CLT-L-F] Central Limit Theorem. http://mathworld.wolfram.com/CentralLimitTheorem.html

[CNVRG] Convergence of random variables. http://en.wikipedia.org/wiki/Convergence_of_random_variables

[Cunningham] Ward Cunningham. Ward Explains Debt Metaphor. http://c2.com/cgi/wiki?WardExplainsDebtMetaphor

[Elsammadisy] Amr Elsammadisy. Opinion: Refactoring is a Necessary Waste. http://www.infoq.com/news/2007/12/refactoring-is-waste

[Fowler99] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley 1999

[Hohmann08] Luke Hohmann. *Beyond software architecture: creating and sustaining winning solutions*, p. 14, Addison-Wesley 2008.

[L-Cond] Lindeberg's condition. http://en.wikipedia.org/wiki/Lindeberg%27s_condition

[Leffingwell10] Dean Leffingwell. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*, ch. 20 Addison-Wesley 2010.

[Lognorm] Lognormal distribution http://en.wikipedia.org/wiki/Log-normal_distribution

[Reinertsen09] Reinertsen, Donald. *The Principles of Product Development Flow: Second Generation Lean Product Development*, Celeritas Publishing 2009

[WikiSRP] Single responsibility principle. http://en.wikipedia.org/wiki/Single_responsibility_principle

Despair Programming

Managing relationships is vital to a successful project. Teedy Deigh looks back on a lifetime's experience.

Agile development has created a culture of newly weds, programmers coupled in pairs oblivious to the fate that awaits them. As with all forms of coupling, the short-term benefits are outweighed by the long-term consequences. The optimism of a new relationship spelled out in code never lives up to the story, no matter how it is prioritised.

There has been much talk and many studies about how effective pair programming is, but clearly all those involved are looking for some kind of meaningful justification that makes sense of their predicament. Apparently pairing improves code quality and is enjoyable, but I doubt that: how can you really have fun and program well when you keep having to remove your headphones to listen to someone else questioning your mastery of code?

Good pairing is supposed to involve alternately navigating and driving. From what I can tell, this means navigating the quirks of another's style and conventions while driving home your own beliefs about how to organise things properly. It is a contest in which there will be a winner and a loser. So much for team spirit!

I suspect that financial debt – which is like technical debt but with money – is a contributory factor. PCs, however, are not that expensive. Surely companies can spare enough money to supply each programmer with their own PC or, at the very least, a keyboard they can call their own? The point of a PC is that it's personal – the clue is in the name. Sharing a computer is like sharing a toothbrush, only more salacious.

For example, the practice of promiscuous pairing is often promoted.

You swing from one partner to the next willingly, openly and frequently. Such loose coupling demonstrates a lack of commitment and sends out the wrong moral message. If you're going to have to pair, you should do it properly, all the way from 'I do' to 'Done'. It is likely that there will be an eventual be a separation of concerns, but that at least avoids the risk of communicating state-transition diagrams and infecting your C++ code with explicit use of the standard library namespace.

One thing that might be said in favour of pairing is picking up new skills. For example, I have learnt to use a Dvorak keyboard and a succession of editors with obscure key bindings and shortcuts. Being able to present new and existing partners with an unfamiliar and hostile environment puts them off their guard and sends out a clear signal about the roles in the relationship. I also find pairing can be effective with newbies. They can either sit and watch for a few hours or they can drive while you correct them from the back seat.

These benefits, however, are few and far between. The day-to-day reality is more cynical: the constant nagging, the compromises you make, the excuses you have to make up, the methods you use, the arguments, the rows, the columns... and you sometimes have to put up with your partner snoring after you've offered an extended and enlightening explanation of some minor coding nuance they seemed apparently unaware of!

So, don't impair to code, decouple. ■

Teedy Dee is resolutely a singleton and pro-singleton. Chris Oldwood once suggested they pair to help her address her singleton issues. Chris assumed her icy stare and menacing silence were some kind of consent, but soon learnt to regret his suggestion and interpretation, coining the term despair programming in the process. Teedy works in a team of one. Chris now works as far away from Teedy as possible.