contents

credits & contacts

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

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

Advisors:

Phil Bass
phil@stoneymanor.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@gmail.com

Pippa Hennessy
pip@oldbat.co.uk

Advertising:
Thaddaeus Frogley
ads@accu.org

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

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

Information and Membership:
Join on the website or contact

David Hodge
membership@accu.org

Publications Officer:
John Merrells
publications@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial: Can We Change For The Better?

I've just read in the 27th August copy of New Scientist about some experiments with "cultural transmission" - one chimpanzee from each of two groups was taught a method of getting food from a complicated feeder. When the groups were allowed to watch, they followed the lead of their respective expert in spite of one method being more efficient. Even when some members of the less efficient group learnt the better way the majority reverted to the original inefficient strategy.

I once worked in an organisation where two teams adopted new development practices and both significantly outperformed the company norm for projects of the sort. One team embraced unit tests as part of the deliverable for each piece of work, the other adopted a strong independent verification culture. (Depending on the metric they were around 60-100% more efficient than would be expected from historical data.) Attempts to introduce practices from either team to the other were singularly unsuccessful – they'd be tried briefly and, even if they worked better, were soon abandoned in favour of the original strategy.

It may seem a long step from experiments on animals to software development, but if Pete Goodliffe can draw comparisons between programmers and monkeys, then surely I can be allowed one between developers and apes. I've seen many attempts to introduce more efficient ways of doing things fail in just this way. Depressing isn't it? (By the way, the New Scientist article is at `http://www.newscientist.com/article.ns?id=dn7913` - it summarises a report in *Nature* and gives the reference as "DOI:10.1038/nature04047".)

I've given an example of the failure to adopt new habits from developing software, but there are others from other aspects of life (without stopping I can think of examples of driving, drinking and housework habits that illustrate the same theme). People, like the other apes described above, learn habits from their peers and then are very reluctant to change them. And groups that have a way of doing things will try to impose them on a newcomer with a better way rather than embracing change.

In the two teams described above there were two mindsets at work – one believed that developers could be trusted to have the discipline to produce working code, the other that developers need to be policed to impose the discipline to produce working code. The incompatibility of these mindsets was demonstrated when management shuffled the team members around to staff new projects. The new projects were far less successful because team members were being asked to work in ways that they knew were less effective than when they'd worked on one of the successful projects. The intended "spreading of new ideas" backfired as each camp demonstrated that the unfamiliar approach was unworkable. When the dust settled, both approaches had been abandoned and the community returned to the comfortable practices that pre-dated these two innovative projects.

Change is hard – I know that. I resist change all the time: I use C++ for some tasks that, for example, Python is more suited to. (The last time was "flattening" a directory structure while avoiding name clashes by incorporating the path elements in the target file names. And yes, there are probably tools that would be even better than Python.) Part of the reason is that I knew immediately how to do it in C++ but would need to look things up in Python (or any other tool) - this introduces unpredictability into the length of the task. But equally, I know that if I were to spend the time learning the tools to do these things that way I'd soon doing them faster.

Sometime in the last year a colleague was having trouble avoiding a lot of duplicate code and asked for suggestions. After a quick explanation it appeared that the problem was deducing an appropriate intermediate type for processing before conversion to a target type. I showed him a trick that introduced the needed "extra level of indirection" (what the cognoscenti would call a C++ template metafunction). He was initially very taken with this technique as it factored out the variability in just the way he was trying to achieve. But after a few days of working with this approach he went back to duplicating the code (with minor variations) for each of the types he was working with. Maybe a template metafunction was wrong in this case and there was a better solution (Adaptor pattern, perhaps), but the point is that he stuck with the comfortable old way of doing things. (And he did want to change – he had asked me for a way to avoid all this duplicate code in the first place.)

I've heard countless stories from developers that have successfully fought against the accepted way of running a project in order to introduce practices they then proceed to demonstrate are better. After which the organisation, for no very clear reason (and

Copy Deadlines

All articles intended for publication in *Overload 70* should be submitted to the editor by November 1st 2005, and for *Overload 71* by January 1st 2006.

with a great sigh of relief), reverts to the "normal" way of doing things. And, in these case, the people involved will frequently have all agreed that the changes should be adopted.

The same report in New Scientist tells another story that indicates that there is hope. This story is of a killer whale that invented a new way of catching seagulls. (For those that are interested in catching seagulls the method is this: he regurgitates some fish onto the water's surface; then he waits below the surface for a seagull to appear and take the fish, whereupon the whale lunges catching the seagull in its mouth.) Whales do not patent such innovations and this idea spread rapidly through the local whale community. Clearly the reward of a tasty feathery snack was enough to overcome any natural reluctance to change. Or perhaps whales are just more intelligent than apes?

Maybe changes in behaviour are successful for reasons that have nothing to do with the effectiveness of the new behaviour? Certainly there is little evidence for many of the "best practices" adopted in our industry (and even less for many "standard practices"). There are plenty of practices in common use whose only merit appears to be that they are in common use and, therefore, one cannot be blamed for using them.

The role of cultural groupings in the adoption of new practices is also significant. An idea will only succeed if adopted by the majority of a group. This must make small groups more susceptible to change – an idea can circulate quickly around a small community and gain acceptance, whereas most of a large community may never become aware of it at all.

A lot of small communities all following their own whim means that a lot of ideas are being tried. My question, therefore, is whether there is any way to tell the good ones from the bad? If this is easy and the good ideas stay, then the division of the C++ community I wrote about in *Overload 67* may be a blessing. But how to tell the good ideas from the bad?

It may sound easy – the ideas that work are good, the ones that don't are bad – but it isn't. Consider the two teams I described above. Both had ideas that worked, both had successful projects that appear to have proved them right. But the ideas didn't work together – indeed the two groups had difficulty finding enough common ground to establish a dialogue.

If you believe developers cannot be relied on to validate their own code then discussion of programmer-written tests is missing the point (and risks both the tests and code making the same error), while if you believe in testing your own work the idea of co-ordinating with an independent tester seems cumbersome (and risks disrupting the flow of work).

And it is rarely as simple as comparing two ideas. There are a lot of ideas in play simultaneously in any software development project. Projects vary in many ways: budgets, technology, etc. Teams also vary in size, experience and skills. In short, a direct comparison between two ideas in software development is rarely possible.

I started with an example of two teams in the same organisation, which eliminates a lot of variability. The teams were of similar size and experience and the projects had comparable time and staffing budgets and were based on the same technology. And even then there was no clear answer – except that by changing their development processes both teams improved on past results.

While these teams both succeeded in producing good results the benefit of this was short term. A larger group, the organisation, reacted in the same way as the apes in the New Scientist story – although a few individuals learnt a better way of developing software, the organisation as a whole reverted to the earlier, less efficient, methods. (And the individuals who wanted to practice what the organisation had enabled them to learn chose to leave.)

Knowing about a problem is not enough, nor is knowing the solution, nor even intending to apply the solution. Change requires a firm intent that doesn't come easily or cheaply, which is why habits are so persistent. Not changing also has a cost. The difference is that change is a one-off cost whereas not changing is a continual drain.

There is a very interesting attempt to change being made by the Commonwealth of Massachusetts. They appear to have realised that the choice of data formats for storing documents is significant. As they have an "open government" obligation to ensure that the information is accessible to members of the public both now and in the future, they are required to care about this.

In order to address this need they have been working with a range of parties in the software to identify a strategy that meets their goals of accessibility to these documents, and have adopted one based on open standards for their document formats. The point of this is that anyone can produce software that works with these formats now, or at any time in the future. And by working with suppliers on the choice of standard they have ensured that suitable software is already available – no doubt more will follow if this initiative survives.

Why do I say "if"? Well, they have a good technical solution to a problem, but that is far from all that is required to make a change. I'm sure that their current principle supplier of office applications would prefer not to have competition. (And Microsoft has considerable influence.)

After decades of suffering incompatibilities between office applications from different vendors, and even different versions from the same vendor, I think a change to a common standard could be a change for the better. The opportunity for change is here, and there has recently been a dramatic demonstration of the costs of a lack of interoperability in FEMA's handling of Hurricane Katrina.

Whales can learn new, improved ways to do things. Can we?

Alan Griffiths
overload@accu.org

Incremental Design:
A Case Study of a Compiler
by Bryce Kampjes

Introduction

Agile development is great for keeping control of details, but how well does it deal with development involving heavy design? This article is about using a very agile approach to write a dynamic compiler. Compilers are probably the best area to do design in because of the large body of compiler literature and theory.

I'll discuss my experience of writing a compiler using an agile approach. The key challenge is working a lot of design, often from papers not personal experience, into an agile development process. Agile incremental development works very well for writing a compiler because it's a very effective way to learn from other people's written experience. It's a very effective way to learn while developing and get the learning into the code quickly.

The compiler, Exupery, is a hobby project. It compiles Squeak Smalltalk bytecodes into x86 machine code which is then executed inside the same process that is running the compiler. One of the original goals was to learn something general about software engineering by personally writing a technically difficult piece of software. The other original goal was to produce a useful open source project. Hopefully, you'll gain some insight from this article.

Compilers

A minimally useful compiler is a large project and a fully optimising compiler doing everything imaginable is impossibly large (a trip to the library or CiteSeer[1] makes it even bigger).

There are several diferent compiler designs that could make sense for an object oriented dynamically typed language. The possible compiler designs run from compiling as quickly as an interpreter interprets [1] [2] to very slow compilers that produce very high quality code. [3]

Most JIT (Just-in-Time) compilers compile fast but not as fast as interpreters, they stick to linear time worst case algorithms because compilation happens at run time where noticable pauses are not acceptable. Most mature batch compilers tend to be very slow, aiming at fast execution times (at least at higher optimisation levels) though a lot are simpler to make them buildable.

JITs are a great strategy for very fast compiling compilers because they can recoup on the second execution. But they are much less appealing when execution time is important because the compile time pauses will get longer (potentially a lot longer, as compilers often use $O(N^2)$ - or greater - algorithms).

Compilers offer some strong benefits for up front design because of the rich body of literature, but they also have some strong disadvantages. It's likely that there are no successful projects working in the same design space: language implementation; optimisation style (fast compilation or fast execution); and basic framework (SSA, dataflow, simple tree walkers, one pass). It is also very likely that no-one on the team has done anything similar.

Judging design literature without having personally implemented similar projects is hard, if not impossible, but many key decisions have to be made early. These are really project-level design, not technical, but they'll influence the choice of algorithm and intermediate forms.

1 http://citeseer.ist.psu.edu/

Register Allocation

Register allocation is the process of assigning all the variables (including ones holding intermediate values) to a fixed number of physical registers. There are several different approaches to register allocation. The main two are colouring register allocators and linear scan allocators. Colouring register allocators are slower but produce better code. Linear scan allocators have seen a lot of work recently because they are more suitable for JIT compilers.

Colouring Register Allocators

```
while hasSpills
  build interference graph
  simplify and move removal
  hasSpills := select registers
  if hasSpills
    insert spill code
  else
    assign variables to registers
```

A colouring register allocator works by first building an inteference graph. Nodes represent the variables, and edges mean the variables can be live at the same time. If two variables have an edge connecting them then they cannot be assigned to the same register.

Simplification is the process of removing nodes where their degree (number of edges) is less than the number of registers available. While simplifying we also remove moves to and from registers where possible (coalescing). As variables are simplified they are pushed onto a stack. If no variable can be simplified then one is chosen and pushed onto the stack as a spill candidate.

Selecting registers involves popping variables off the stack then assigning them to a register that isn't used by any of their neighbours. It will be possible to find a register for it because when it was selected it had less neighbours than the number of registers . If a variable was pushed as a spill candidate then we try and find a register for it if some of its neighbours were allocated to the same register but if we can't it will be spilled (stored in memory rather than a register).

Colouring register allocation was first practically formulated in 1981 by Chaitin [5]. Briggs introduced a few key improvements in 1992 [6] including making final spill decisions during selection, which is more efficient because some of a variable's neighbours might have been allocated the same register, so spilling would have been overly pessimistic. George and Appel in 1996 [7] extended Briggs' work by coalescing moves while simplifying, rather than as a separate phase, which allows a lot more moves to be removed. There's a lot more work on colouring register allocators – Google scholar finds over 500 documents. It would be stupid not to do a serious literature search before implementing in a domain which has been studied so much.

Squeak and Why It's Challenging to Implement Efficiently

Squeak is the main open source implementation of Smalltalk. Smalltalk is a pure dynamically-typed object oriented programming language. Everything is an object including Contexts (stack frames), Blocks (closures), and integers. `SmallIntegers` fit into a machine word with a tag and will overflow into large integers.

The core language is very pure. Even conditionals `ifTrue`: and `ifFalse`: (the equivalent of `if` statements in C) are implemented in the library, but Squeak, and most other Smalltalks, cheat here for speed. `SmallIntegers` are only slightly different from other objects - it's possible to add methods to SmallIntegers but not to

subclass them. Squeak, and most other implementations, provides special bytecodes that perform `SmallInteger` operations. If the arguments are not `SmallInteger`s or the operation fails then a normal message `send` will be performed.

Every expression needs type checking, which is the cost of being a pure dynamically typed language. This costs about 2 cycles to type check per operand. Theoretically type checking can be removed in most cases.

Tagging and detagging integers in Squeak is slower than necessary because the integer tag is `1` rather than `0`. Small integer operations need to remove the `1` tag before operating then add it afterwards.

Booleans are `real` objects. The only restriction is there can only be one true object and one false object in the system. This means that the expression `a < b ifTrue: [do something]` naively needs to perform the comparison, convert the control flow into a boolean object (a little `if then else` sequence), then convert the boolean back into control flow (another `if object is true then else` sequence). That's a lot of instructions and also an extra conditional branch for the processor to (potentially) mispredict.

`Sends` (function calls) are very common. Theoretically all expressions are `send`s and most are in practice. `While` loops, `if` expressions, and small integer operations are theoretically `send`s but are expanded when generating bytecodes.

Smalltalk runs in an image which hosts both the IDE and the program being developed. The entire system is written in itself and modifiable live. Smalltalk systems host their own development tools. It's possible, and normal, to change how the development system works live during normal development. The debugger, profilers, and editors (called browsers) are normal code that can be inspected and changed.

Compilation is done incrementally. When developing, individual methods are compiled as they are edited and will be used for every call after they have compiled. It's also possible to modify the system programmatically. This is how the programming environment is implemented. Any method may be modified at any time including by code. Any object may be swapped with any other object at any time (`become:`), all references to the objects get changed by this operation. A lot can be changed during normal program execution.

Squeak is currently implemented by an interpreter. The interpreter executes a bytecode in about 10 clocks which is the branch misspredict penalty. That's a fairly tough target to beat with a naive compiler but a well designed simple compiler could do it.

Initial Planning

When starting a large project, the first problem is deciding if the project is worthwhile, which involves enough design to estimate the costs of a minimally useful system. A minimally and generally useful compiler is probably around 10-20kloc which is a large system to write as a hobby. It's possible to write a compiler in less, but to be worth the cost of maintenance it must be noticably faster than the interpreter. Simple compilers can be slower than tuned interpreters. Exupery is a big project for a single person's hobby, so ideally it should also have big motivating goals. The end goal is to be as fast as C or Fortran for most programs.

Exupery is designed to become faster than any previous Smalltalk implementation by combining a solid classical optimiser with inlining driven by dynamic type feedback. That's too much to achieve in one step but it still helps to know that the end goals are possible.

To completely eliminate the cost of Smalltalk's dynamic power we need to:

1. Remove integer tagging and detagging.
2. Remove boxing and deboxing of Booleans and Floats
3. Remove all send overhead including calling blocks (higher order functions)
4. Optimise the low level intermediate to an equivalent level to C

This could be done by:

1. Having solid back end producing good code. In practice this just means a good register allocator and some care. This was the biggest source of remaining waste noticed in Self [4]

Smalltalk Syntax

Smalltalk syntax is very simple. There are three types of messages, blocks, and literal expressions.

`Factorial printString`	Unary messages are a single word.
`* + ** <=`	Binary messages look like operators in other languages. They have an argument before and after them.

`Receiver printOn: 'hello'.`
`Receiver`
`  at: 'a key'`
`  ifAbsent:`
`  [self doSomething]`
	Key word messages are a sequence of words ending in :. Each argument always has a keyword before it ending in :

`[1+ 20] [:each| each + 10]`	Blocks are bits of code that can be executed later. They look like `[anExpression]` or `[:each| self aMessage: each]`. They can take arguments, each argument begins with a `:`, after the last argument there is a `|`.
`|a b c|`	Defines three local variables called `a`, `b`, and `c`. Definitions can only occur at the start of methods or the start of blocks.

Precedence is unary messages then binary messages then keyword messages. So `aStream nextPutAll: 10 * 200 factorial` fully bracketed is: `aStream nextPutAll: (10 * (200 factorial))`. The result is to print a very large number on `aStream`.

Control flow is logically implemented using message `send`s and blocks. `a < b ifTrue: [self doSomething]` executes the block `[self doSomething] if a < b. [a < b].whileTrue: [a := a + 1]` is a basic `while` loop. It first executes `[a := a + 1]` while executing `[a < b]` returns `true`. `#at:ifAbsent:` used as an example of a keyword message will execute its block argument if it couldn't find the key (the first argument). Reading `[]` brackets as if they were brackets from C is tolerable but Smalltalk's `[]` are semantically richer because they return a first class object.

Smalltalk is a natural environment for JITs (which it's had since '83). Writing a traditional JIT inside the language poses problems. A JIT normally stops execution until it has compiled a method then jumps into it. What happens if it's written and running in the environment it's compiling and needs to compile part of itself to finish compiling a method?

2. Using dynamic type feedback to discover what types are used for each `send` site then inlining commonly called messages.
3. Using type analysis to remove unnecessary type checks and boxing/ unboxing of Booleans and Floats
4. Induction variable analysis will allow the removal of most type checks of loop counters and array range checks

Dynamic type feedback provides information on what types were actually used. The solid optimisation framework can then remove redundant tagging and detagging in the common case code[2] and moving unnecessary code out of loops (e.g. the code that figures out how big an object is to range check it. The array is almost certainly the same object across the entire loop).

So long as compilation happens in a background thread and never causes execution to halt, it's possible to combine very fast generated code with no pauses. Background compilation also makes it possible to run the compiler in the same environment that's being compiled. The advantages are: there are no compilation pauses, slow optimisations can be used, and the compiler can be written in Smalltalk as a normal program. The key disadvantage is there must be another way of executing Smalltalk but we already have an interpreter.

Task Breakdown

The vision of a compiler above is too big to commit to building. So let's break it down into three different projects. First the full compiler meeting the initial ambitious goals. This is the goal that's really motivating. Second the simplest possible compiler that would be useful. This is the first version that should get widespread use. Third the simplest possible compiler that can be built and tested. This compiler is the first stepping stone to the second project.

The project's current high level breakdown (release plan) is:

1 Get it compiling a basic program. Just to compile an iterative factorial
2 Make that faster than the interpreter.
3 Compile the bytecode benchmark in the same process as the interpreted compiler
4 Make that faster than the interpreter
5 Add support for most of the language. (minus blocks)
6 Optimise `send`s with PICs
7 Optimise bytecodes. Make it faster than VisualWorks, the fastest commercial Smalltalk
8 Make it practical. This is the current task
9 Make sends fast. Full message and block inlining
10 Build a basic optimiser. Just to remove integer tagging and untagging
11 Extend the optimiser to handle classical optimisations such as common sub expression elimination and code hoisting. Moving `#at:` overhead and write barrier checks out of loops is a specialised case of code hoisting which may double the bytecode performance
12 Extend the optimiser down to replace the low level optimiser. This will allow the classical optimisations to remove redundancy that was hidden inside the tagging and boxing code.
13 Add floating point support. To be worthwhile, native compiled floating point support needs to remove most of the boxing and deboxing of floats. This may require the entire optimiser in this plan or it might be worthwhile just with tree traversal based optimisations and dynamic type feedback. Measurements on real code could answer this question.
14 Induction variables. Extend the optimiser to optimise induction variables (a fancy name for loop counters and friends). This should allow all range checks using looping variables to be fully optimised and to remove the overflow checks when incrementing loop counters. It should remove the last remaining overhead in common array loops.

The original release plan was for five major releases 1) a basic compiler, 2) decent bytecode performance (at least twice the interpreter's) and compiling inside the image, 3) add most of the bytecodes, 4) decent `send` performance (probably inlining), 5) make it practical. Each release is a few months' full time work.

Implementing the release plan up to "Make it practical" or "Make sends fast" would produce a roughly minimal compiler that was practically useful. This is where the compiler is up to now, it's four times faster for the bytecode benchmark and twice as fast for the `send` benchmark than the interpreter. The only algorithm that's more complex than a tree traversal is the colouring register allocator. That such a simple compiler could be so much faster than the interpreter was easy to predict initially by looking at the performance of traditional JITs, which cannot use any complex optimisations because they cannot afford long compile time pauses.

From the minimal practical compiler, I broke out a minimal testable compiler which was the first thing built. The minimal testable compiler just compiles an iterative factorial method into assembly which was then assembled and linked to a C driver routine. The next iteration added a register allocator and some performance tuning to instruction selection to use some addressing modes. Then I started compiling directly to machine code and running the generated code inside the same process that was running the compiler.

This little compiler could then be extended by adding language features and optimisations. It is the core of the minimal useful compiler. The overall architecture is there, all the phases exist in a nontrivial form. Building up towards the basic minimal compiler is now just work and keeping control of the details. It's a recursive process that's re-entered in the section "Later Design".

Breaking down large tasks is critical. There are too many details to keep straight. Stuff that isn't important at the strategic level because it definitely can be made to work can still cause crashes or wrong answers. Making a task breakdown is often enough design to safely begin implementation.

At the beginning a brief project plan is useful to plan just enough to see that the project is worth starting, however, justifying a large project by its end goals is a mistake. Ideally each iteration should pay for itself. [9]

Testing and Small Steps

The main benefit of tests is the ability to work in very small chunks. It's possible to write (or modify) tests one evening then modify the code later. It's very easy to pick up coding on a broken test.[3]

Exupery is written using two separate test suites; each has almost complete test coverage. The programmer tests are unit tests with some

2 Common case might just mean any code path that's ever been called. There is a lot of code generated for cases that are not expected to happen but theoretically might. For eample, the integer addition code includes a full message send just in case the arguments are not integers or the addition overflows. Moving type checks and tagging out of the common case code and into the return from an uncommon send is a simple and effective optimisation, it just needs the level of abstraction.

3 This is vital for hobby projecs where 2 full days in a row is rare (there's always something else to do in a weekend) and very nice for professional work where distractions from coding are often unavoidable (live customer problems, meetings, whatever).

code integration tests thrown in, they will never crash the development environment. The customer tests compile and run example fragments[4] which can crash the development environment if they generate faulty programs (programs that corrupt memory etc).

When writing code to extend the language coverage, first I write some customer tests while figuring out what all the special cases are for the new statement. Then I'll run them, they break. Then it's time to write programmer tests to drive the implementation. The programmer tests drive the implementation of the language feature for each component. For a new integer operation this will start with byte code reading, then intermediate generation, then adding the new instruction to the back end. One the tests have been written, it's possible to focus on the details without needing to keep the entire picture in mind all the time, if a critical detail is ignored the tests break. Tests specify behaviour precisely enough to allow it to be safely ignored, this reduces the amount that I need to keep in my head, which allows me to work with smaller chunks of time.

Just having a customer test suite that compiles and runs programs is not enough. There are too many ways to implement, the test that drives development should provide the direction to develop. The test will often be written one evening, and the code a few days later, therefore the test must carry the subtlties to be implemented because short term memory cannot.

Debugging and Reliability

Debugging can be a major problem because bugs usually show up when you run a compiled program and it crashes (or produces the wrong answer). The first stage is figuring out why the generated program is crashing. Then figuring out where the bug is in the compiler, which is non-trivial because one stage may be failing because a previous stage did something unanticipated.

Reliability is a big problem when writing a compiler, there is a lot of functionality, and every user visible action involves most of the compiler, including several complex algorithms . Thus tracking down a bug can be a lot of work, because first it must be found in the compiled code, then it must be traced back though the compiler. Even when compilation fails it's often not clear which stage is at fault, is the register allocator failing because it's complex and still buggy, or is the input invalid because of an upstream bug?

Reliability is a key issue for compilers, not just because it's better not to have real compiler bugs, but also because it's easy to lose control of quality. They are too complex to debug into shape if the quality ever drops.

Non-deterministic bugs are surprisingly easy to create. First because analysis algorithms may only exhibit a bug on one ordering (iterating over hash tables can make order nondeterministic) – the order isn't important for correctness, but it can expose or hide bugs. Having a large automated test suite and running it frequently makes intermittent bugs obvious as the suite will only sometimes fail or crash.

Large test suites help to ratchet up the quality. By keeping all the tests passing it's easy(ish) to avoid introducing new bugs. Adding more optimisations however adds more ways for subtle bugs to creep in as there are more ways for features to interact. Optimisations require more customer/integration tests to get the same level of coverage.

Originally I thought that I'd need to run my acceptance tests in a separate forked image to make it safe. The customer tests run generated machine code which can crash the development environment. Having your development enviroment which includes the editors and debuggers crash regularly during development is painful. Forking separate processes hasn't been necessary, manually identifying the crashing tests from the stack trace when it crashes and throwing an exception at the beginning of the crashing tests has proved enough. It's very unusual for a lot of tests to be broken for a long time. The programmer tests keep the quality high enough that unexpected customer test crashes are sufficiently rare.

It's important to be able to quickly fix broken tests. The code would get very brittle because it would take too long to debug if any test started failing. Knowing that all the tests passed recently reduces the amount of code that could have introduced a bug and thus the time required to fix it.

A Little Duplication

Often there isn't a clear right way to factor (erm, design) some code but it is painfully clear that the current structure is inadequate. Sometimes there are obvious small improvements that we can make but there are still cases where there isn't an obvious way to improve the structure. There it's worthwhile to introduce some duplication.

A common case is test code, what should vary between the tests? What should be factored into helper methods and hidden? There's a cost, more context is needed to read the method, if it's reused this is a big saving. Copying and pasting the first test 10 times, changing the parts needed, then refactoring away the duplication has worked well.

Intermediate creation and assembling are good examples in the production code, both have an involved detail driven design which evolved through refactoring. The initial duplication provided the environment to find the right structure, then refactoring consolidated the winning design features.

One big problem with intermediate creation was finding a design that was both clean and easy to test without duplication in the tests. The obvious way to test intermediate creation is to compile short methods into intermediate then check that the intermediate generated is correct. The problem is the basic intermediate building blocks such as integer tagging are repeated, making them very hard to change without needing to change a lot of tests.

My first approach was to break the intermediate generator into two objects, the first handling the higher level logic and the second handling the repeated low level operations. This helped, but there was often too much code in each test. The next insight was to make the intermediate emitter mock itself by having an instance variable that normally just impersonated self (this in C++). This provides a flexible recursive way to test intermediate emission. A message can be sent either to self (included in the test) or to the mockable instance variable (which is normally the same as self). This provides enough flexibility to test any part of intermediate generation in isolation. Testing components in isolation requires very decoupled design.

A little duplication is often a good thing, especially when there isn't an obviously right answer and a better design is obviously needed. The duplication allows the code to evolve in different ways without being overly constrained with the requirements from elsewhere. Duplication and copy and paste programming makes experiments with design cheap. Refactoring will remove the duplication later, once the kernel of a design has been found. Tests over the duplicated code makes it safe to refactor once a decent structure is found. Refactoring away the duplication is likely to refine the design as it needs to generalise to deal with more solutions.

4 There are also performance tests, but their use is separate, they do not check correctness.

Refactoring and Architectural Correctness

Large refactoring often involves moving temporarily away from architectural correctness.

For instance, there are two different, and both correct, ways of generating expressions in a simple intermediate language. Either expressions are written in trees representing the abstract syntax tree, or individual sub-trees are written out sequentially as a list. Eventually the code will end up as a sequential list, but the tree form provides a lot of cheap and simple manipulative power (the tree optimisers). Either has obviously correct semantics but mixing them arbitrarily will produce bugs when side effecting expressions are executed in a different order.

Who is responsible for unique execution of expressions is a similar problem. The expression `array at: anExpression` uses `anExpression` multiple times. First to type check it (is it an integer), then to lower range check it (is it greater or equal to 1), then to upper range check it, then finally to index into the array. Is `anExpression` responsible for returning a register in case it's used multiple times or is `#at:` responsible for placing the result in a register before using it multiple times?

In both cases I've switched from one architecturally correct style to the other. The simplest way to work is to generate code adding it directly to a basic block and make each expression responsible for returning a register which is given to any expression which receives its result. This is very easy to test – check that all results from expressions are either a literal or a register. However tree optimisations require expressions in trees so they can optimise operation sequences across bytecodes (high level tree optimisations). Also it's slightly more optimal to make the user responsible for uniqueness because then larger trees are formed to feed into instruction selector.

The problem is, when refactoring from one style to another, there will be a noticeable time when the entire system is broken. Either format in both cases is correct but half and half is always wrong although it may work sometimes. But to refactor, we should be moving in small steps keeping a working system. Breaking the system provides a lot of tactical guidance (fix what's broke until everything works again).

Here, there's a key difference between having strong guarantees and having all tests pass. The strong guarantee is the reason to believe the current tests are sufficient. If all tests are passing and they are good enough to provide a strong guarantee then the program should be bug free. Finding strong guarantees is definitely a heavy thought activity, closer to Dijkstra than TDD is normally portrayed. An architectural refactoring will often need the the test suite to be changed, not to make it pass again, but to make it cover the new design well enough.

Later Design

Design after the project has started is building the understanding needed to make decisions when required. It's best to make major decisions by deferring them to the last possible moment, but that's the wrong time to build the understanding required to make them.

Later design involves exploring how different infrastructure investments produce different performance improvements. The tactical decisions will be driven by specific benchmarks, but choosing which benchmarks (and tests) will drive is important. Also looking for powerful combinations that enable a lot of later features to be easily implemented is important.

When key decisions need to be made early it pays to make them by default, say by ignoring the cost of moves and registers knowing a colouring register allocator can clean this up. It's important to do the thought experiments to know that there are good ways to solve the problems when they come up.

Many design decisions are easy to make at the right time but some are not. The best that I can do is guess which decisions need thought then start thinking and reading about them preferably well before they really need to be made.

Often the key long term decisions are attempts to find optimal and minimal sets of optimisations – to find the different hills. Key questions worth thinking about are:

- What is the minimal infrastructure needed to compete with C in speed for most programs?
- What is the minimal infrastructure required for high speed floating point and what is required for a useful floating point speed improvement?
- What is the minimal system that is practically useful and worth the risk of using for real production use for any major market segment?

There are some features that are worth investing in because they will take us to a better hill. Dynamic inlining is the ideal solution to common message sends as it makes common sends nearly free. Building a decent optimisation framework will make a lot of serious classical optimisations very cheap to build. Often key pieces of infrastructure can not be justified by their first use (register allocation could be) but are still worth building.

Register Allocation, the First Big Piece of Infrastructure

Why choose to use a colouring register allocator?

1. Removes complexity from the front end
2. Hides two address operations from the front end which both simplifies the front end and makes it more portable
3. Provides an efficient general solution
4. Not optimal for the x86 without using very sophisticated solutions because of register pressure due to only having 6-7 usable registers, however, move coalsecing is very useful to avoid the front end needing to know about 2 operand addressing.

I chose to implement a naive register allocator first then evaluate whether it was worthwhile to use a complex colouring allocator. I also made design decisions heavily favouring a colouring register allocator by ignoring the cost of moves and registers in the front end. The rest of the code was designed to work well with a colouring register allocator but debugged with a very simple allocator. This meant that when writing the allocator it was much easier to debug because the rest of the initial compiler was written and tested. Delaying the final decision to implement meant that when I did implement I was sure it was necessary rather than just relying on a (correct) guess.

The colouring register allocator was the first, and currently only, complex algorithm in Exupery. It was in the initial design but not implemented until the second iteration. The first iteration compiler was very slow, it assumed that there was a good register allocator that would remove unnecessary moves and efficiently allocate the remaining registers. The colouring register allocator was needed to make Exupery faster than the interpreter.

The register allocator serves to hide most of the oddness of the x86 from the compiler's front end. It cleans up the moves introduced

to fake 3 operand instructions and removes most of the short lived temporaries used to by general code sequences in the front end.

At what stage is it justified to introduce a complex register allocator? The choice is either a single complex register allocator or adding a lot of complexity to the front end of the compiler to minimise the use of registers and move instructions generated.

Design Lock In, Assumptions Can Spread

Good design is keeping each decision in its own module so that each design decision can be changed independently. Unfortunately, a very usefully made decision can leak as the rest of the system starts to rely on it.

A separate version of the `#at:` primitive is now compiled for each receiver. This enables some code to be calcalated at compile time rather than run time. This means that the versions of the `#at:` primitive that are used for getting characters out of strings are compiled into different methods to the versions used to get objects out of Arrays. Squeak only uses one primitive for all common `#at:` calls.

Compiling each form of `#at:` efficiently is fairly easy but creating a single implementation that deals with all cases is hard. A case statement would be needed for the type of the receiver which adds to the overhead. As the compiler inlines calls to `#at:` it would mean that every call would have code for every kind of optimised `#at:` implementation.

Compiling a different method for each receiver also allows the compiler to calculate the offset where indexed variables start. Objects in Squeak start with some named instance variables then may have indexed (array) instance variables afterwards. This way a simple array can be implemented as a single object, the named instance variables contain any bookkeeping the class needs, and the indexed instance variables contain the array data. It's a space optimisation from the '70s, changing it would be a lot of work and would also make it much harder for people to use Exupery.

The array `#at:` primitive in Squeak looks in the class to find out how many named instance variables the class has, looks at the object to find out how big it is, range checks, then does the lookup. By specialising for each receiver class we can avoid all overhead from having a variable number of named instance variables. The primitive code can also ignore all the other implementations of `#at:` in the system because they are sent to other classes, a bytecode implementation needs to deal with this type checking.
However, having a separate compiled representation for each receiver type opens up the possibility of inlining self sends with no run time overhead, because the receiver type is known at compile time. As more code starts to rely on the receiver type being known at compile time then this decision will slowly get locked in.

The register allocator has been locked in by a similar process. The front end assumes that registers and move instructions are free. Switching to a simplier register allocator would involve a lot of rework for the rest of the compiler to avoid generating wildly inefficient code.

How Should Compiled Code Interact with Objects?

Compiled code often needs to refer to objects in garbage collected object memory (the image). This is tricky because objects may be moved on a full garbage collect as the garbage collector will compact memory.

The current solution is naive but it works. There is an array of objects which the VM has a pointer to. Compiled code accesses objects indirectly through this array. This way there is only one pointer that needs to be coordinated with the compacting garbage collector. It does add an extra level of indirection to every object access. These accesses happen during the method entry code and also during an inlined primitive.[5]

Ideally, it would be nice to be able to hard code a pointer to an object in the compiled code. However if there is a garbage collect then the object may have been moved leaving a dangling pointer, unless something is done. There are two options, either update all the pointers to objects in compiled code, or discard all the code in the code cache (flushing it). Code can be regenerated dynamically, so throwing it away is a simple and viable solution to managing dependencies.

If there weren't pointers from objects into the code cache then discarding all the code would be a very simple operation. Unfortunately `CompiledContexts` (stack frames) contain pointers into the code cache which make it very quick to perform returns. The pointer is the machine code return address. There is also a dictionary (hash table) used to look up compiled methods. Another option to speed up sends and returns would be a send cache [2] which is more complex but the contents could easily be discarded.

Luckily, Squeak objects have a few common classes encoded directly into the header in a 5 bit field. This means that type checks will be fast for common operations such as arrays. Having three different encodings for classes (`SmallIntegers`, compact classes using the bit field, and classes requiring a pointer to the class) is unfortunate when implementing fast `send`s or general type lookups.

My first cut simple approach of using indirection through an array turned out to be good enough especially with a simple optimisation relying on the compact class encoding. This will make it a little harder to eliminate compact classes (it could be done in a few lines of Smalltalk). A better solution may be needed sometime in the future but there is no urgency.

Making Exupery Practical, the Current Design Problem

Making Exupery practical is the current major strategic problem. The basic requirements have been met, Exupery is four times faster for the bytecode benchmark and two times faster for the send benchmark. Most of the implementation is solid and the last few pieces of scaffolding (early implementations that were too naive to be left in but allowed other parts to be built properly) have been removed.
The goal is to provide a good useful speed improvement for normal code. Fast benchmark performance is nice, but it doesn't guarantee a useful speed improvement for general code especially when only a few benchmarks are used. That a lot of the inner loop methods have been rewritten in Slang[6] or C doesn't help, because it removes a lot of the easy methods which Exupery could have optimised.

The issues stopping usefulness are bugs, driving the compiler, and missing features. When compiling code outside of the test suite it's still too common to run into bugs, to be generally useful it must be trusted, preferably to compile anything at any time. Driving dynamic inlining manually is not pleasant, you need to specify what `send`s will be inlined and with what receivers. There are also probably a few missing features, for example, blocks (closures)

5 A send to some primitive operation that isn't implemented in Smalltalk, either because it is a basic operation or for performance.

6 Slang is the language that the Squeak interpreter is written in. It is a cross between Smalltalk's syntax and C's semantics. One advantage of Slang is it's possible to debug the interpreter in Squeak before compiling down to C. It also provides a clean way of writing primitives to speed up slow loops from code that was originally Smalltalk.

might be needed, or support for more of the super extended bytecodes (bytecodes that have an extended argument field).

There are a host of small issues, including bugs when compiling things that have never been compiled before, and small features that will really matter if the compiler is running in a background thread automatically compiling methods. For instance, compiled code does not listen for interrupts, so it can not be aborted and compiled code can not currently be single stepped.

Most of these issues are small, the key decision is figuring out what needs to be done for the compiler to become practical. That's best done incrementally by fixing the current most obviously limiting problem until it isn't a significant problem. These kinds of problems are best solved with plenty of feedback by fixing one then re-evaluating.

Inlining and Polymorphic Inling Caching

`Send` performance is critical to high performance Smalltalk. One current goal is improving the general `send` performance. This is building towards full dynamic inlining.

Dynamic inlining is strategic, because it changes both how the compiler is used and also the overall cost structure. Dynamic inlining makes common sends free, this means a lot of optimisations to work around slow sends can be removed.

The phases are:

1. Polymorphic Inline Caches
2. Specialised Primitives
3. Inlined Specialised Primitives
4. Inlined General Methods

The breakdown was to get a useful speed improvement as early as possible and also to make the individual sections as short as possible.

Polymorphic inline caches provide two things, first they provide a fast form of `send`s, second they provide a way to get dynamic type information. Dynamic type information may be required to drive inlining[7].

Specialised primitives are compiled versions of a method containing a primitive. They are compiled for each different receiver type, this allows the compiler to generate custom code for each implementor. They need PICs to make the sends fast enough to compete with local implementations.

While having fast primitives with fast sends to them is nice, it's still not as quick as a specialised version of the message (say `#at:` and `#at:put:`). One way to speed them up further would be to inline the primitives. Specialised inlined primitives are very nice because they provide a generalised framework to speed up many primitives to the same levels that the basic integer operations enjoy. The type feedback means that your method dynamically gets the right primitives inlined rather than a specific one chosen by the VM implementor. It's also a building block towards full inlined primitives.

Specialised inlined primitives was the last thing added to the breakdown. They change the shape of the breakdown because they remove the need for PICs to get inlined primitives working. There's still a practical issue that non-inlined primitives will be slower, compiling shouldn't make code slower, even temporarily (at least until compilation is automatic). The idea of specialised inlined primitives didn't occur to me until after I'd implemented PICs and specialised primitives. Insight is nice but it doesn't necessarily come at the right time.

It could be argued that it would have been better to go straight for profile driven inlining rather than using PICs. With hindsight, and original foresight, just using inlining might have been better than beginning with PICs, because inlined primitives were needed to regain enough `#at:` performance. This would have involved finding a way to inline `#at:` methods without using PICs to get type information, instrumenting a compiled non-PICed specialised primitive might have been possible. The problem is it would have been much slower after the first compile but before inlining. Whether this would have been tolerable is hard to say (without having dynamically driven inlining to perform real empirical experiments)

Design as Reading, Writing and Discussion

A lot of initial design work involved reading around the field, there's a lot to know - from modern CPU architecture to the two main compiler approaches that it's based on (Self and classical/SSA optimising compilers).

Moving into new areas also will bring on a lot of design by reading. Adding a solid optimiser will involve hitting the books. My current feeling is using SSA from the beginning makes sense, it's not that much more complex (and probably simpler including a basic optimiser) than building a traditional dataflow analyser.

Documentation is something that you shouldn't do too much of on an agile project because demanding documentation increases the cost of change. Agility is producing a low cost of change instead of guaranteeing a perfect result first time. But some is helpful.

One group of documents that's always worth having is those that pay for themselves in writing. Writing is a good way to regain the big picture. Writing can be a good way to have a conversation without anyone to talk to, it's a form of documentation that's more valuable for single person projects than larger projects where there are other people to talk to.

Open source projects also have interesting documentation requirements. The teams are dispersed with different goals and sources of funds (often people's hobby and education time). Being developer driven, documentation often suffers (though commercial development is often undocumented for very similar reasons). But being made up of geographically and organisationally diverse groups, having decent documentation is a serious asset. A key reason is to minimise the amount of time needed to come up to speed on a project. There are a lot more people who have a few weekends available than a few months. But there have to be sub-projects available that could be done by a new person in a few weekends.

Critical reading is essential, the only way to work effectively is to build from experience in literature. There is the trap of implementing something that only works on paper and not in code. Writing is very useful for a single person project because it replaces conversations with peers who are knowledgable about the projects specifics.

Conclusion

Both theory and incremental design are invaluable tools when writing a compiler. Incremental design provides a great way to understand how to apply theory, and also the means to escape from theory's traps where things work brilliantly, but only on paper.

Thinking and doing at the same time, or very closely interwoven, is

[concluded at foot of next page]

7 It can also be obtained by profiling if the profiler records a method and its receiver as well as its sender. this may not be enough for some primitives which is partially why I originally decided to implement PICs.

Vorsprung Durch Testing

by Kevlin Henney

At times it might seem as if the T in TDD stands for Trendy, but there is more to Test-Driven Development than just a statement of fashion. There is also more to it than just testing.

It is possible to identify a subset of three motivating practices in TDD that characterise a fairly conventional and uncontentious form of unit testing [1]: *programmer testing responsibility*, *automated tests* and *example-based test cases*. These form a unit-testing base that can be employed in the context of both static and agile development macro processes, and were motivated and demonstrated previously on the humble but surprisingly rich example of a sorting function in C [2]. Thus, programmers are responsible for unit testing their work, with system-level testing a separate and complementary role and activity; tests should be executed automatically — execution of code by code — rather than manually; tests are black-box tests expressed as specific examples of typical or edge cases of using the unit under test.

The next step is to recognise that effective testing can be more than just bug hunting. In TDD unit testing helps to support and drive design, and vice-versa. Three more practices can be identified that build on the core unit-testing foundation to provide us with a micro-process component that also supports design: *active test writing*, *sufficient design* and *refactoring*. These design-focused practices expand the role of the basic unit-testing practices: examples drive the scope of design [3], programmer responsibility extends to the suitability and quality of code over time — not just at a single point in time — and automation underpins the practical execution of this approach.

Active Test Writing

Black-box testing by example is not just limited to exploring the correctness of an implementation against an interface contract: it is also useful for framing and presenting it, and for formulating and exploring the contract itself. In other words, design.

Passive testing is essentially the process whereby the feedback of tests is limited to defect detection. Tests are typically written some time after the code they test, where they play what is essentially a destructive role: they cannot confirm total correctness, only the presence of incorrectness. Although such an approach to testing has obvious value, it can encourage an approach to both design and testing that is overly formal and sequential. The opportunity to learn about what is being designed, and how to design it better as a whole, is missed [4]. Defects lead to localised fixes, but the test-writing process does not influence the key decisions in a design, which in effect is considered frozen. The feedback loop is too long, so there is less motivation to change things because of the feeling of "what's done is done". The code has effectively gone into conventional maintenance mode early, even though initial development may be ongoing.

Active test writing adopts a more balanced perspective, using the act of test writing as a creative exercise to balance the more destructive intentions of test execution. Tests represent a first point of use of an interface, and the ease or difficulty of writing test cases gives instant feedback on the qualities of the interface and the implementation behind it.

High coupling manifests itself in tests that are difficult or — in simple unit-testing terms — impossible to write. For example, an object that depends on data that could be passed in, but has instead ended up being coupled to a configuration file or registry, a database connection or some global variable (whether expressed obviously as such a variable or disguised as a singleton object).

Low cohesion manifests itself in supernumerary test cases that test quite unrelated features, suggesting that inside a given unit there are smaller units struggling to get out. For example, the standard C `realloc` function expresses three quite distinct behaviours: `malloc`, `free` and, err, `realloc` [5]. The standard `java.util` package contains miscellaneous unrelated facilities — collections, event-handling models, date and time handling, internationalisation features... and further miscellaneous miscellanea. It also stands as a caution to anyone who might consider `util`, `utils`, `utilities`, `utility`, etc, to be a clear and cohesive name for a header, a package, a library, etc.

In terms of organising the *active* part of *active test writing*, there are many options. The bottom line is that writing of test code is carried out in close proximity — both space and time — to writing of production code. The writing of test cases and corresponding

---

[continued from previous page]

a critical technique when working on software in an area with a lot of design literature, especially when lacking first hand experience.

Using a simple solution to a sub-problem at a minimum enables everything else to be debugged separately to the complex solution. Often the simple solution is enough.

Breaking large problems down into smaller ones that can be solved individually is vital to keep the risks low [8] and also because, often, the experience of building the component is required to understand how to design it.

The major goals of the project were valuable for providing consistency and focus, but have yet to become useful when justifying the time spent so far because they are too far away. The short term rewards have been enough to make it worthwhile but they do change, at the beginning working on a hard problem really deepens understanding, as the project progresses community membership becomes more important.

Bryce Kampjes
bryce@kampjes.demon.co.uk

References

1 May, C. *Mimic: a fast System/370 simulator*., 1987 Symposium on Interpreters and Interpretive techniques.

2 Deutsch, L. Peter, Schiffman, Allan M. *Efficient Implementation of the Smalltalk-80 System*, Principles of Programming Languages ? 1983

3 Appel, Andrew, *Modern Compiler Implementation in C*, 1998, Cambridge University Press

4 Holzle, Urs, *Adaptive Optimization for Self: Reconciling high performance with Exploratory Programming*, 1994, PhD Thesis

5 Chaitin, M.A., Register allocation via coloring, 1981, *Computer Languages* 6, pp. 47-57

6 Briggs, P., *Register Allocation via Graph Coloring*, 1992, PhD from Rice University

7 George, L. and Appel, A. W., *Iterated register coalescing*, 1996, ACM Trans. on Programming Languages and Systems 18(3), pp. 300-324

8 Henney, K., *Stable Intermediate Forms: A foundation Pattern for Derisking the Process of Change*, Overload issue 65 February 2005

9 Beck, K. and Fowler, M., *Planning Extreme Programming*, 2001, Addison-Wesley

"Big", as in "a Lot of", not just "a Bit of"

It is worth clarifying what BUFD (or BDUF, as it is also known) entails, because this appears to be an occasional source of confusion. For example, misunderstanding its meaning can lead to proclamations such as the following [9]:

> *I can't tell you how strongly I believe in Big Design Up Front, which the proponents of Extreme Programming consider anathema. I have consistently saved time and made better products by using BDUF and I'm proud to use it, no matter what the XP fanatics claim. They're just wrong on this point and I can't be any clearer than that.*

And, to demonstrate the point, Joel Spolsky makes available for download a so-called functional spec of a commercial product, codenamed Aardvark. However, the deeds do not support the words. The document may have been written up front, but hunt all you like for big design because you won't find it. Strong belief and pride appear to have clouded correct use of accepted terminology.

The accepted archetype of BUFD arises from the strict waterfall approach of defining development as a precisely phased pipeline of activities, so that requirements analysis strictly precedes design activity, which strictly precedes coding, which strictly precedes testing. In a bid to reduce risk from unknowns later in the lifecycle, a BUFD approach doesn't just do a bit of design up front, it does a lot. Hence the use of the term *big* rather than *a bit of* or *some*. The BUFD path is paved with good intentions — even if somewhat suspect — but the idea is that the design goes into a lot of detail, specifying internal structure to the nth degree — from packages and classes right down to private methods and private data. In essence, a blueprint that supports a plan-driven model of development.

However, at the beginning of the Aardvark spec is the following note:

> *This specification is simply a starting point for the design of Aardvark 1.0, not a final blueprint. As we start to build the product, we'll discover a lot of things that won't work exactly as planned. We'll invent new features, we'll change things, we'll refine the wording, etc. We'll try to keep the spec up to date as things change. By no means should you consider this spec to be some kind of holy, cast-in-stone law.*

So, of all the things this spec might be, a big, up-front design document is not one of them. It makes this quite clear to the reader by describing itself as "a starting point for the design" not "the design". Reading further into the spec uncovers frequent use of words such as "maybe", "probably" and "possibly" to describe certain technical decisions. And then there is the length of the document itself: twenty pages. When you strip away the extraneous details, such as the front cover, preamble and the neo-Hungarian coding conventions, you are left with a shorter document that outlines some of the core requirements, proposes a user interaction model and sketches a few features of the architecture. The document is also not heavy on text and is fairly generous with its use of spacing. Whichever way you look at it, this is not big design. Which all comes as a welcome relief, but does rather undermine the claim of its author.

Advocates of genuine BUFD would regard the Aardvark spec as incomplete and insubstantial, lacking detailed specifications of code structure or the look and feel of the application. They would tar it with the same brush that the article uses to daub XP. I believe that the contrast the article is trying to make is to compare no up-front design with some up-front design, not with big up-front design. Joel Spolsky is actually advocating a design approach based on sufficiency, exploration and incrementalism. So although he may not be on the same page as XP advocates, he is many pages short of being a fully paid-up BUFD practitioner.

implementation can be interleaved, with one following or preceding the other closely, or stepped a little further apart. Being able to write a test case first is a useful and helpful discipline, but only dogma would suggest that its exclusive use is an absolute requirement and a necessary prerequisite of TDD. However, although writing test cases much later than the target code can work, both the quality of the feedback and the motivation to do so is weaker.

Sufficient Design

This continuous and reflective view of design at the code face may raise another question in some minds about the whole nature of developing iteratively and incrementally: why not just "do the right thing first time"? Perhaps surprisingly, I have heard this question posed as a serious criticism, but the question itself raises more questions about the meaning of the question and the questioner's assumptions than it does about agile development techniques at any level. It assumes that the "right thing" is in some way knowable "first time" and constant thereafter. However, the "right thing" is dependent on time and is anything but constant, so both "right thing" and "first time" lose their simple interpretations. The learning nature of software development pretty much guarantees that the knowledge of what it is to be built and how it can be built are moving targets. While they may not necessarily be wild and erratic, their variability stands to undermine any approach that is based on constancy and precognition. The difference between a process with no variables and one with some is the difference between defined and empirical processes. Treating an empirical process as a defined process is a problem waiting to happen [6].

Yet there can still be a lingering sense that sorting everything out up front is both reasonable and do-able, leading one way or another to a *big up-front design* (BUFD) phase (see sidebar, *"Big", as in "a Lot of", not just "a Bit of"*). This inevitably leads to overdesign. Design based on assumptions that turn out to be incorrect needs to be reworked, often quite late. Design that tries to tackle uncertainty by being less specific becomes lost in technical detail focused on generality rather than on the actual problems that need to be addressed. At the opposite end of the spectrum is *no up-front design* (NUFD), which represents a failure to exercise, in a timely manner, even the most basic knowledge about what is to be developed. An approach based on a view that accepts change but seeks stability is likely to be a more reasoned one, albeit a little rougher in its detail up front, where roughness implies sketched rather than shoddy. An approach based on what I have referred to in recent years as *rough up-front design* (RUFD) can steer this middle path. Establish a stable baseline architecture that expresses a common vision and a sketch of what is to be worked on, without wasting time on details that are better expressed and handled in code or that are best left until more concrete knowledge is available. Note that *stable* is not the same as *static*, so the architecture is open to change rather than being frozen. This approach can also be dubbed *sufficient up-front design* (SUFD).

Sufficient design in TDD manifests itself in test-bounded design increments, where tests describe the scope of what is being worked on at any point in time. This moderates creeping featurism, cuts extraneous code and encourages incremental and measured progress. Active testing supports the goal of sufficient design by keeping the role of functions, classes and packages

clearly defined. Tests bound the functional behaviour of these units, keeping them 'honest' with respect to their current role in the enclosing system.

Driving the design from the baseline architecture through tests leads to more cleanly separated units with a close dependency horizon (a dependency occurs where one unit, e.g. class or header, depends on another unit for its definition, e.g. inheritance or inclusion, and the dependency horizon for a given unit is where its dependencies end, i.e. where its immediate dependencies, and their immediate dependencies in turn, and so on, have no further dependencies). Of course, there needs to be some coupling at certain levels otherwise, by definition, no coupling results in no system.

Refactoring

Mrs Beeton's Victorian domestic advice [7] is surprisingly relevant to modern code:

> *A dirty kitchen is a disgrace to all concerned. Good cookery cannot exist without absolute cleanliness. It takes no longer to keep a kitchen clean and orderly than untidy and dirty, for the time that is spent in keeping it in good order is saved when culinary operations are going on and everything is clean and in its place. Personal cleanliness is most necessary, particularly with regard to the hands.*

This is the very motivation and essence of refactoring. Refactoring preserves the functional behaviour of a piece of code while changing — and, one hopes, improving — its developmental qualities. Refactoring is a stable and local change, typically motivated by a required change in functionality. Operational behaviour, such as performance or memory usage, may change, but improvement of operational qualities rather than developmental qualities is the focus of the similar but distinct activity of optimisation.

Changes to functionality may follow the line of the existing code easily, requiring no more than a consistent extension or in-place modification of the code. At other times a change in functionality may also suggest a change in implementation of an interface. An existing implementation may be OK in other respects, but may support the functionality change poorly, requiring undue effort to implement it. For example, the need to perform general date arithmetic on an existing date representation that favours presentation over calculation, such as `YYYY-MM-DD`, suggests that a change in representation may be appropriate before extending the functionality [8]. Alternatively, the quality of an existing piece of code may generally be poor, caught in a tangle of spaghetti flow or spaghetti inheritance. For example, a self-aware class hierarchy, where the root of the class hierarchy depends on other classes in the hierarchy, can be a troublesome knot in the dependency graph of a program, rather than an exemplary pattern to be followed elsewhere.

Refactoring acknowledges that we can lay down code in confidence but still learn better ways of achieving the same end. Indeed, it is more than this: the learning is not simply passive; it is put into practice and draws from practice. Of course, there is a risk that making such a change is not necessarily an improvement: any modification runs the risk of introducing a bug. Therefore, practise with a safety net: refactoring should be undertaken with a clear head, with another pair of eyes, with tools, with tests, or with any suitable combination of these. In the context of a test-driven approach, test cases offer a regression test suite that act as a baseline for both refactoring and optimisation.

Given that the inevitability of change is one of the few constants in software development, this active acknowledgement and positive support of change through tests is reassuring. Refactoring is the other side of the design coin from what we might consider to be *prefactoring*. Refactoring adjusts the design vision and detail after the fact to balance the formulation beforehand.

Test Match Report

Test-Driven Development is a bar-raising, learning process. Removing the tests leaves the safety net at ground level and knowledge localised, isolated and transitory. A TDD approach offers more than just a pile of tests: it offers specification as well as confirmation. Both of these reasons are sufficient to justify writing tests that sometimes apparently test the trivial. And specifying even the trivial to be sure that it always works means that regression testing comes for free as part of the deal.

Another consequence of TDD is the resolution of an imbalance in the traditional view of testing. Testing is often characterised as a destructive activity, and one that is predominantly quantitative in its feedback. TDD makes testing a constructive activity, with qualitative feedback on design, not just defect reports.

TDD is not a total process: you need other complementary drivers to move development forward. For example, an incremental macro process where each increment is scoped with respect to functional or technical objectives provides a good backdrop to the code-facing emphasis of TDD. Likewise, practices such as reviewing, joint design meetings and continuous integration support and are supported by TDD. It is also important to distinguish TDD from XP: although historically it emerged from XP, TDD is neither a synonym nor a metonym for XP. Implementing XP necessitates employing TDD, but the converse is not true. TDD fits with many different macro-process models. There are many more programmers practising TDD in other processes than are using it in a strict XP environment.

kevlin@curbralan.com
kevlin@acm.org

References

1 Kevlin Henney, "Driven to Tests", *Application Development Advisor*, May 2005, available from `http://www.curbralan.com`.

2 Kevlin Henney, "C-side Re-sort", *Overload 68*, August 2005.

3 Brian Marick, *Driving Software Projects with Examples*, `http://www.exampler.com`.

4 Kevlin Henney, "Learning Curve", *Application Development Advisor*, March 2005, available from `http://www.curbralan.com`.

5 Kevlin Henney, "No Memory for Contracts", *Application Development Advisor*, September 2004, available from `http://www.curbralan.com`.

6 Ken Schwaber and Mike Beedle, *Agile Development with Scrum*, Prentice Hall, 2002.

7 Isabella Beeton, *Mrs Beeton's Every-Day Cookery and Housekeeping Book*, Ward, Lock & Co Ltd, 1872.

8 Kevlin Henney, "The Taxation of Representation", *artima.com*, July 2003, *http://www.artima.com/weblogs/viewpost.jsp?thread=8791*.

9 Joel Spolsky, "The Project Aardvark Spec", August 2005, `http://www.joelonsoftware.com/articles/AardvarkSpec.html`.

Polynomial Classes

by William Hastings

A multivariate polynomial in several variables, for example, $7 + 4x^2y^3 – 5yz^6$, may be thought of as a polynomial in z with coefficients that are themselves polynomials in x and y. These coefficients, in turn, are polynomials in y with coefficients that are polynomials in x. How can we implement this recursive definition in C++? In this installment and the next we will explore two distinct solutions. In the first solution the number of variables is built into the definition of the class. In the second, the number of variables may vary at runtime. Before proceeding, note that similar issues arise if we define a multi-dimensional array as an array of arrays (which is useful if the number of non-zero entries is relatively small).

The Straightforward Approach

If we know the number of variables at compile time, then the overall structure of a possible solution is straightforward. Suppose we define a template class

```
template <typename C> class polynomial {...};
```

where `C` is the type of the coefficients. This class is a container of pairs of the form [power, coefficient]. For the polynomial above, we have two pairs, $[0, 7 + 4x^2y^3]$ and $[6, –5y]$. A polynomial in three variables with numerical coefficients of type `double` has type

```
typedef class polynomial<polynomial<polynomial
   <double> > > poly3_type;
```

A useful way to define `poly3_type` is:

```
typedef class polynomial<double> poly1_type;
typedef class polynomial<poly1_type>
   poly2_type;
typedef class polynomial<poly2_type>
   poly3_type;
```

We can write

```
variable_list<char> vars3("xyz");
poly3_type p(vars3), q(vars3);
p.read("5-3x^2y^4+x^3z^3");
std::cin >> q;
std::cout << "p has " << p.number_terms()
   << "terms.\n";
std::cout << "p+q=" << p + q << '\n';
```

Note that the caret (^) means exponentiation, so `y^3` means y*y*y. The role of `class variable_list` will be explained below.

Problems With This Approach

This approach works well, but there are a few problems to overcome. First, suppose we want our polynomial class to have a method that returns the number of terms. In our example polynomial there are three terms. This is easy to compute – just sum the number of terms in each coefficient:

```
template <typename C>
int polynomial<C>::number_terms() const
{
  int count = 0;
  for (const_iterator it(begin());
   it != end(); ++it)
    count += it->coefficient().number_terms();
  return count;
}
```

Here, the method `coefficient()` returns a `const` reference to the second element of a pair in our container. The problem is that this code will not compile. Note that the call to `number_terms()` inside the `for` loop is not a true recursive call. For example, if we call this method for a polynomial of type `poly3_type`, then `it >coefficient()` has type `poly2_type` and so we are calling `poly2_type::number_terms`, which in turn calls `poly1_type::number_terms`. It is this last call that causes the problem, since now `it >coefficient()` has type `double`, and type `double` has no method named `number_terms`.

There are at least three ways to deal with this problem. First, we could use partial specialization to handle the end case. Having defined

```
template <typename C> class polynomial;
```

we can now define, for example,

```
template <> class polynomial<double>;
```

But this requires specializing `class polynomial` for each possible numerical type, for example,

```
template <>
   class polynomial<std::complex<double> >;
```

A better way is to turn this approach on its head and define the partial specialization for the case where the coefficients are themselves polynomials:

```
template <typename C>
   class polynomial<polynomial<C> >
{ . . . };
```

With this approach, we have

```
template <typename C> class polynomial {
//Coefficients are numeric
public:
  int number_terms() const {
    int count = 0;
    for (const_iterator it(begin());
       it != end(); ++it)
      if (it->coefficient() != 0)
        ++count;
    return count;
  }
  //other methods . . .
}
```

and the partial specialization

```
template <typename C>
   class polynomial<polynomial<C> > {
//Coefficients are polynomials
public:
  int number_terms() const {int count = 0;
    for (const_iterator it(begin());
       it != end(); ++it)
      count +=
         it->coefficient().number_terms();
    return count;
  }
  //other methods . . .
}
```

Using a Traits Class

The objection to this approach is that we must repeat all of the code in the definition of the class and in its partial specialization,

even if the code is identical. A variant on this approach is to use a traits class[1]. We define (for numeric coefficients)

```
template <typename C> class coefficient_traits
{
public:
  static int number_terms(C const &c)
  { return c != 0 ? 1 : 0; }
  //other methods . . .
};
```

As above, we need to specialize this for the case of a multivariate polynomial:

```
template <typename C>
class coefficient_traits<polynomial<C> >
{
public:
  typedef polynomial<C> coef_type;
  static int number_terms(coef_type const &c)
  {
    int count = 0;
    for (coef_type::const_iterator
       it(c.begin());
       it != c.end(); ++it)
      count +=
         it->coefficient().number_terms();
    return count;
  }
  //other methods . . .
};
```

Then we can define `number_terms()` in `class polynomial` by

```
int number_terms() const {
  int count = 0;
  for (const_iterator it(begin());
     it != end(); ++it)
    count += coefficient_traits<C>::
       number_terms(it->coefficient());
  return count;
}
```

Of course, the traits class can be used to record other properties. Suppose we have a template class called `numeric_traits` that includes all the methods required for coefficient_traits with definitions that are appropriate for most numeric types. Then we define `coefficient_traits` by

```
template <typename T>
class coefficient_traits
: public numeric_traits<T> { };//Default case
```

Of course, we must provide a partial specialization for coefficients that are polynomials:

```
template <typename C>
class coefficient_traits<polynomial<C> > {...};
```

We can also specialize as needed for numeric coefficients. For example, to output the polynomial x – 3y, we need to know that the coefficient of y is negative. (Otherwise, we would output x + –3y.) Now for a coefficient `c` of type `double`, we can ask if `c<0`, but not if `c` has type `complex`. Therefore, we include in `coefficient_traits` a method

```
static bool is_negative(C const &c);
```

For `C` of type `std::complex<T>`, we can override this method in an appropriate way.

One possibility is:

```
template <typename T>
class coefficient_traits< std::complex<T> >
: public numeric_traits<std::complex<T> >
{
public:
  static bool is_negative(coef_type const & c)
  { return c.imag() == 0 && c.real() < 0; }
  //other overrides
};
```

Numeric versus Polynomial Coefficients – a Third Way

Let's get back to our original problem, namely how to differentiate between coefficients that are polynomials and those that are numeric. Our third approach relies on the fact that in a template class, a method that is not invoked must not be instantiated. With this approach the method `number_terms` calls one of two methods depending on the coefficient type. For this to work, the choice must be made at compile time. To this end we use a technique described in Alexandrescu's book [2]. First an auxiliary class will be used to distinguish the two cases:

```
template < bool b > struct Bool2Type { };
```

Now, define in `class polynomial<C>`

```
int number_terms() const
{
  return number_terms(
    Bool2Type<(nbr_vars==1)>());
}

int number_terms(Bool2Type<false>) const
{
  int count = 0;
  for (const_iterator it(begin());
      it != end(); ++it)
    count += it->coefficient().number_terms();
  return count;
}

int number_terms(Bool2Type<true>) const
{
  int count = 0;
  for (const_iterator it(begin());
      it != end(); ++it)
    if (it->coefficient() != 0)
      ++count;
  return count;
}
```

Depending on the type C, exactly one of the latter two methods will be invoked; the other will not be instantiated, so there will be no compile-time error. The compile-time constant `nbr_vars` has other uses as we will see below. It is defined by

```
namespace detail
{
  //Count the number of variables in a poly
```

```
template <typename C>//Base case
struct var_count
{
  static const int nbr_vars = 0;
};
template <typename C>
struct var_count<polynomial<C> >
{
  static const int nbr_vars =
     1 + var_count<C>::nbr_vars;
};
}
```

and then in `class polynomial<C>`:

```
static const int nbr_vars =
  detail::var_count<polynomial<C> >::nbr_vars;
```

In `class polynomial` most of the methods do not depend on the coefficient type; for the few that do, I have used both the `traits` class and method pairs using `Bool2Type`.

Variables

Before proceeding with how `class polynomial` is constructed, we need to consider variable names and polynomials with different numbers of variables. The representation of polynomials we are examining here implies an ordering of the variables. For example, we may think of p as a polynomial in y with coefficients that are polynomials in x. Of course, we could just as well think of p as a polynomial in x with coefficients in y. What is essential, of course, is consistency. To compute p + q , for example, we want p and q to have variables in the same order. Furthermore, to simplify input it is useful to know in advance what the variable names are.

Let's put the responsibility for maintaining a collection of variable names with order in a separate class

```
template <typename NameType>
   class variable_list;
```

This class is just an ordered container of names of type `NameType` (e.g. `char`)[3]. Its primary constructor is

```
variable_list(char const * vars);
```

where the character string `vars` might be `"xyz"`. The constructor will transform `vars` into an ordered list. (By default, this is reverse alphabetical order.) So, for example,

```
variable_list<char> vars1("xyz");
```

and

```
variable_list<char> vars1("zyx");
```

are equivalent.[1]

Now, to construct polynomials we have

```
typedef variable_list<char> var_list_type;
var_list_type vars3("xyz"), vars2("xy");
poly2_type p(vars2);
poly3_type q(vars3);
```

Polynomial Arithmetic

Suppose we want to compute `q -= p`. (We use subtraction as an example since p – q <> q – p , and so it is slightly more complicated than addition.) Mathematically, q – p makes sense as a polynomial whatever the variables of p and q. It will have three variables if the variables of p are also variables of q. For example, if p = x + z and q = xyz, then q – p is x + z – xyz. On the other hand, if p = w + x, then we have a problem with q –= p, since q – p now has four variables while the number of variables of q is fixed at three. (We'll see how to solve this problem in part 2.)

In any event, I would like to be tolerant and allow p and q to have different variables, or at least a different number of variables. This implies that we need to declare the subtraction operator by

```
template <typename C1, typename C2>
????? operator -(polynomial<C1> const & p,
                 polynomial<C2> const & q);
```

The fun begins with the return type. If `C1` and `C2` are not the same, the return type should be the type of the argument that has more variables. For example, the return type of `p - q` (of types `poly2_type` and `poly3_type`, as above) should be `poly3_type`. Here are two ways to deal with this problem. Both approaches need to compare the number of variables of types `polynomial<C1>` and `polynomial<C2>`, so let's encapsulate this information in a class:

```
namespace detail
{
template
<typename C1, typename C2, bool b>
struct pick_poly //case b is false
  { typedef polynomial<C2> type; };

template <typename C1, typename C2>
struct pick_poly<C1, C2, true>
  { typedef polynomial <C1> type; };

template <typename C1, typename C2>
struct compare_coefs {
  static const bool use_c1 =
    (polynomial<C1>::nbr_vars >=
     polynomial <C2>::nbr_vars);

  typedef typename
  pick_poly<C1,C2,use_c1>::type return_type;
};
}
```

Here the constant `use_c1` is true if `polynomial<C1>` has at least as many variables as `polynomial<C2>` and `compare_coefs<C1,C2>::return_type` is either `polynomial<C1>` or `polynomial<C2>`, depending on which has more variables. Hence this type should be the return type of `operator -`. Our first approach to defining subtraction uses `Bool2Type` defined above:

```
namespace detail {
  template <typename C1, typename C2>
  polynomial<C1> do_subtract(
   polynomial<C1> p,
   polynomial<C2> const & q,
   Bool2Type<true>)
  {
    return p -= q;
  }
```

1. This default behaviour is easy to change.

```
  template <typename C1, typename C2>
  polynomial<C2> do_subtract(
   polynomial<C1> const & p,
   polynomial<C2> const & q,
   Bool2Type<false>)
  {
//create a polynomial of type polynomial<C2>:
    return -q += p;
  }
}
template <typename C1, typename C2>
typename detail::compare_coefs<C1,C2>
              ::return_type
operator-(polynomial<C1> const &p,
          polynomial<C2> const &q)
{
  return detail::do_subtract(p, q,
    Bool2Type<detail::compare_coefs<C1,C2>
                  ::use_c1>());
}
```

The third argument to `do_subtract` ensures that we return a polynomial of the correct type. Note that if p and q always have the same number of variables, then the second version of `do_subtract` will not be instantiated. Also note that the types of the first argument in the two versions of `do_subtract` are slightly different: the first takes a polynomial by value, the second by `const` reference. To see why, examine the bodies of the two versions.

Using `enable_if`

The second approach uses `boost::enable_if`, which is based on the SFINAE [5] principle. Two versions of `operator -` are defined, but only one version is valid:

```
template <typename C1, typename C2>
typename
boost::enable_if_c<detail::compare_coefs
   <C1,C2>::use_c1, polynomial<C1> >::type
operator -(polynomial<C1> p,
  polynomial<C2> const & q)
{
  return p -= q;
}
template <typename C1, typename C2>
typename
boost::disable_if_c<detail::compare_coefs
      <C1,C2>::use_c1, polynomial<C2> >::type
operator-(polynomial<C1> const &p,
          polynomial<C2> const &q)
{
  return -q += p;
}
```

Suppose `use_c1` is true. Then in the second version of `operator -`, `disable_if_c<…>::type` is undefined, while in the first version, `enable_if_c<…>::type` is its second template argument, i.e. `polynomial<C1>`. By SFINAE the second version is not considered by the compiler when it looks for an appropriate candidate for `operator-=` (because the return type does not make sense). If `use_c1` is false, then the first version is invalid and in the second, `disable_if_c<…>::type` is its second template argument, i.e. `polynomial<C2>`.

Handling Different Variables

We are now ready to consider how to deal with polynomials in different variables. Before proceeding, note that our representation of a polynomial implies an ordering of the variables. The polynomial p= x + y + z, for example, is a polynomial in z with coefficients that are polynomials in x and y. These coefficients, in turn, are polynomials in y with coefficients that are polynomials in x. Of course, we could have used any other order. (For example, we could represent p as a polynomial in x with coefficients in y and z.) To the extent possible, our code should not make assumptions about this order. One assumption we will make, however, is that if two polynomials p and q appear in an arithmetic expression, then the ordering of the variables in p and q agree. For example, suppose p has variables z, x, and w (in this order) and q has variables z, y and x. The ordering of the variables in q must have z before x. (The variable y, however, could be first, second or third.)

Let's look at the implementation of the member function

```
template <typename C>
template <typename C2>
inline polynomial<C> &
polynomial<C>::operator -=
        (polynomial<C2> const & q) {
  //. . .
  return *this;
}
```

If the variables of q are a subset of the variables of p, then computing `p-=q` is straightforward. We can relax this requirement somewhat: if q is not constant with respect to some variable, then that variable must be a variable of p. If, as in the last paragraph, p has variables z, x and w and q has variables z, y and x, then q must be constant in y (i.e. no term of q contains a non-zero power of y).

Now, there are two ways to handle the case where p and q have different variables. In the first case, we assume that what must be true is true:

```
template <typename C>
template <typename C2>
inline polynomial<C> &
polynomial<C>::operator -= (polynomial<C2>
const & q) {
  //Compute p -= q
  if (this->first_variable() ==
      q.first_variable()) {
    //subtract two polys in same (first)
    //variable:
    // . . .
  }
  else if (q.is_constant_in_first_variable())
    *this -= q[0];
  else {
    (*this)[0] -= q;
  }
  return *this;
}
```

Some explanation is in order. Let's suppose that the first variable of q is z. The method `first_variable`, returns a name, so `q.first_variable()` returns 'z'. The expression `q[n]` is the

coefficient of the n^{th} power of z (because z is the first variable of q), so `q[0]` is the constant part (with respect to z) of q. If q is constant in z, then `q` and `q[0]` are the same polynomial, so `*this -= q[0]` makes sense in the middle branch.

The last branch is more subtle. Since q is not constant in its first variable z, the variable z must be a variable of p, for otherwise this subtraction must fail. But p and q do not have the same first variable, so the first variable of p is not a variable of q. (It is here that we are using the assumption that the order of the variables of p and q are compatible, as described above.) In short, the subtraction will work in this case only if the variables of q are variables of `p[0]`.

What happens if our assumptions do not hold? That is, what happens if `q` is not constant in some variable that is not a variable of p? The key is to examine the call in the last branch. If `p` and `q` have three variables (e.g. have type `poly3_type`), then `(*this)[0] -= q` invokes `operator -=` with arguments of types `poly2_type` and `poly3_type`, which in turn will invoke `operator -=` with arguments of types `poly1_type` and `poly3_type`. In this last invocation, `(*this)[0]` has type `double`. In order for this to compile, we need to define something like the following. (The actual signature is a little more complicated since the numerical coefficients may have type other than `double`.)

```
template <typename C> inline
double & operator -= (double & x,
                      const polynomial<C> & q)
{
  if ( ! q.is_constant())
    throw std::domain_error(
        "Different variables in subtraction");
  x -= q.leading_constant();
  return x;
}
```

Here is where the error is detected. The method `leading_constant` returns the numerical coefficient of the first term (i.e. the "leading" term). When `q` is constant, this method returns that constant.

Code Explosion

There is a serious problem with this approach. Even though our implementation of `operator -=` may seem compact, in fact it forces the instantiation of several versions of this operator. For example, if p and q both have type `poly3_type` (i.e. polynomials in 3 variables), then fifteen versions of `operator -=` are instantiated. For example, the statement

```
(*this)[0] -= q;
```

means that `operator -=` is instantiated for arguments of types `poly2_type` and `poly3_type`. We saw above that the statement

```
*this -= q[0];
```

invokes `operator -=` with arguments of types `poly3_type` and `poly2_type`. In short, we have every pair of arguments of polynomial type with 0 to 3 variables (except 0 and 0, i.e. `double operator-=(double,double)`.) All of this for just one of the basic arithmetic operators to handle cases that may never arise!

There is a remedy for this combinatorial explosion of code. We are requiring that the result of `p -= q` be a polynomial with the same variables as p. This is possible only if q can be expressed as a polynomial in the same variables as p. In C++ terms this means that we can construct a polynomial that equals q mathematically but has the same type and variables as p:

```
template <typename C>
template <typename C2>
inline polynomial<C> &
polynomial<C>::operator -= (polynomial<C2>
   const & q)
{
  if (first_variable() == q.first_variable())
  {
    //subtract two poly in the same (first)
    //variable: . . .
  }
  else if ( ! q.is_zero()) {
    try {
      polynomial<C> qq(build_variable_list(),
         q);
      //Compute p -= qq . . .
    }
    catch (construction_error const &) {
      throw std::domain_error (
      "Different variables in subtraction");
    }
  }
  return *this;
}
```

The key element here is in the call to the constructor inside the `try` block. The method `build_variable_list()` returns the variables of `*this`. The constructor creates a polynomial with these variables and then attempts to copy the terms of q into this polynomial. If q has terms involving nonzero powers of some other variable, the attempt fails and a `construction_error` exception is thrown. With this approach we do not get the cascading calls to distinct versions of `operator -=` (assuming that the user does not try to subtract polynomials of different types.)

Conclusion

This polynomial class (and two other versions) are available online at `http://www.fordham.edu/mathematics/hastings/polynomialclasses/`.

The class described in this article is called `fixed_poly<C>` to emphasize that the number of variables is fixed at compile time. It is defined in `FixedPoly.hpp` and is available at this web address. Also at this address are several other necessary include files and one source file.

In the next installment, I examine how we can modify the approach described here to allow the number of variables to vary at run time.

William Hastings
<hastings@fordham.edu>

References

1 Alexandrescu, Andrei and Herb Sutter *C++ Coding Standards*, item 65.
2 Alexandrescu, Andrei *Modern C++ Design*, pp29ff.
3 For details see `http://www.fordham.edu/mathematics/hastings/polynomialclasses/var_list.htm`
4 See `http://www.boost.org/libs/utility/enable_if.html`

A Framework for Generating Numerical Test Data

by Peter Hammond

Abstract

While attempting to bring the benefits of early unit testing to a highly numerical application, we found a need to generate large quantities of test data using several independent variables. We found the obvious approach using nested loops unsatisfactory. An alternative was developed using chained objects to represent the independent variables, with minimal repetition of either code or structure.

Introduction

The work described here was part of the development of a highly numerical component within a subsystem of a large (>$200M) defence system project. The component uses Kalman filters [1] to assist in target tracking. The Kalman filter algorithm is a method for reducing noise on measured data, and also allows for interpolation and extrapolation of measurements. The component also does a considerable amount of other processing related to the tracking loop. Not surprisingly for a project of this size, the main project management processes are rooted in high-ceremony, document-centric methods. The data processing subsystem was developed in Ada, using the Rational Unified Process. The development environment used ColdFrame [2] to generation the framework from UML, and encouraged early unit testing using the AUnit unit test framework, an Ada version of the well known xUnit family [3].

We developed this component with a view to using as agile a process as possible within the constraints of the overall project policies. In particular, we wanted to explore the claimed benefits of early testing methodologies such as Test Driven Design, although automatic code generation was used for the structural framework of the component. To reconcile the two, we used a test-first strategy to implement the method bodies within the generated framework. Some parts of the component contain a great deal of numerical code, which poses challenges for this style of testing; there may be many input combinations that need to be tested to give confidence that the function's implementation will perform correctly across the entire input domain. For testing those parts, the following procedure was adopted:

1. One engineer implemented the function in C++.
2. This implementation was used to generate a file containing inputs and outputs for the function, covering as many odd cases as possible (maximum and minimum values, zeroes and angles such as $\pi/2$).
3. Another engineer implemented an AUnit test harness that uses this data.
4. The second engineer implemented the function in production code, directly from the specification without reference to the initial implementation.
5. If the two implementations do not agree within suitable tolerances, allowing for rounding errors, there is most likely a bug in at least one of the implementations. We then entered a debugging phase, as far as possible retaining the independence of the two implementations.

While this procedure does not completely guarantee the correctness of the code, it does provide a valuable check, and a regression test to prove that nothing changes later. The highly numerical code was also Fagan inspected to further add confidence. We see the two approaches as complementary; both approaches found some defects in code that had been through the other.

The Problem

The original test data generators used a simple structure of nested for loops. An example is shown in Listing 1, which creates solutions of quadratic equations using the well known formula. This is of course not a real example; the real algorithms that were being tested are not available for publication. While the nested loops were easy to implement at first, this approach suffered from a number of drawbacks:

- Indentation quickly used up screen space, reducing readability.
- It was necessary to ensure manually that header and data columns match up in the output file, which introduced duplication of structure.
- Code to show progress was repeated by copy and paste throughout the numerous dataset generators, which is clearly duplication of code.
- Housekeeping (managing the loops and files) and performing the calculation were not separated.
- Amongst other faults, this design is therefore breaking two well-known principles in good software: separation of concerns, and avoiding duplication.

```
void generate_quadratic ()
{
  FILE* f = fopen ("quad.txt", "w");
  fprintf (f, "a,b,c,x1,x2\n");
  const int max_column = 30;
  int column = 0;
  for (double a = 0.1; a <= 5.0; a += 0.1)
  {
    printf (".");
    if (++column == max_column)
    {
      printf ("\n");
      column = 0;
    }
    for (double b = 10.0; b <= 20.0; b += 0.1)
    {
      for (double c = 0.0; c <= 5.0; c += 0.1)
      {
        double z = sqrt (b * b - 4.0 * a * c);
        double x1 = (-b + z) / (2.0 * a);
        double x2 = (-b - z) / (2.0 * a);
        fprintf (f, "%.16lf, %.16lf, %.16lf,"
                 "%.16lf, %.16lf\n",
                 a, b, c, x1, x2);
      }
    }
  }
  fclose (f);
}
```

Listing 1: A simple example using for loops

A more elegant design was indicated, one which would:

- Remove duplication, both of code and of structure.
- Automate as much as possible, including progress reporting.
- Separate management of the loops from the business of calculating the test values;.
- Support linear and logarithmic iterations.
- While being minimally intrusive in the implementation of the equations being tested.

Dead Ends

The problem of iterating several functions with the same values could easily be solved by providing the iteration as an algorithm, and passing it the function to be called with the iterated values. An example of this is shown in Listing 2. It would clearly be straightforward to extend this to variable limits and steps by passing in {step, min, max} tuples for each variable. It seems intuitive that this could be extended to arbitrary numbers of iterated values, and probably logarithmic iterations, using template meta-programming techniques. However, despite some considerable head-scratching I could not find a suitable solution along this route. This is not to say that such a solution does not exist, but an easier alternative was found first.

```
// Supply test values to f.
void generate (FILE* out, void (*f)(FILE*,
               double, double, double))
{
  const int max_column = 30;
  int column = 0;
  for (double a = 0.1; a <= 5.0; a += 0.1)
  {
    printf (".");
    if (++column == max_column)
    {
      printf ("\n");
      column = 0;
    }
    for (double b = 10.0; b <= 20.0; b += 0.1)
    {
      for (double c = 0.0; c <= 5.0; c += 0.1)
      {
        (*f)(out, a, b, c);
      }
    }
  }
};
// output test values and results.
void quadratic(FILE* out, double a, double b,
               double c) {
    double z = sqrt (b * b - 4.0 * a * c);
    double x1 = (-b + z) / (2.0 * a);
    double x2 = (-b - z) / (2.0 * a);
    fprintf (out, "%.16lf, %.16lf, %.16lf,"
         "%.16lf, %.16lf\n", a, b, c, x1, x2);
};
const char*
   const quadratic_hdr = "a,b,c,x1,x2\n";
int main()
{
    FILE* out = fopen("quad.txt", "w");
    fprintf (out, quadratic_hdr);
    generate(out, quadratic);
    fclose(out);
};
```

Listing 2: A possible solution using an algorithm and function separation.

A Solution

Being a "classic" C++ programmer at heart, it seemed natural to look for an object based solution, using some kind of object to represent iteration over a range of values, and for these to be chained together to create the nesting by functional recursion. The problem was that the values of all the iterated variables have to be available to the final function that will create the output values.

The first successful form of the framework used objects of a `Value_Iterator` class as members of the client class to do the iteration, with operator `double()` to access the values. The name Iterator may not have been the best choice, as it perhaps implies too strongly the iterators of the standard library collections. However, it does iterate over the input values for the test. `Value_Iterator` is derived from the `Iterator_Base` interface, which defines the methods needed to take part in the recursion. The client of the

```
class Quadratic_Generator : private
Iterator_Manager
{
public:
    Quadratic_Generator () :
        a (0.1,  10.0, 0.1, "a", this),
        b (10.0, 20.0, 0.1, "b", &a),
        c (0.0,  10.0, 0.1, "c", &b)
    {};
    void start ();
private:
    void iterated_function ();
    Iterator* get_head () {return &c;}
    FILE* f;
    Lin_Iterator a, b, c;
};
void Quadratic_Generator::start () {
    f = fopen ("quad.txt", "w");
    get_head()->
       write_chained_descriptions (f);
    fprintf (f, ",x1,x2\n");
    start_iteration();
    fclose (f);
}
void Quadratic_Generator::iterated_function ()
{
    double z = sqrt (b * b - 4.0 * a * c);
    double x1 = (-b + z) / (2.0 * a);
    double x2 = (-b - z) / (2.0 * a);
    get_head()->write_chained_values (f);
    fprintf (f, ",%.16lf, %.16lf\n", x1, x2);
}
int main (void) {
    Quadratic_Generator().start();
}
```

Listing 3: Example of usage of the first attempt.

framework is a class derived from an abstract base, `Iterator_Manager`, which encapsulates common behaviour for managing the recursion. Two concrete `Value_Iterator` classes are declared apart from the manager, to provide linear or logarithmic steps. The framework is shown in the UML diagram figure 1, and an example of its usage is shown in Listing 3.

The `Iterator_Manager` is responsible for managing a chain of `Value_Iterator` objects, although the objects themselves are members of its client specialisation, here `Test_File_Generator`. It provides some functions for management, but it is still abstract, as it requires the client class to define the iterated function and the head of the chain. Its `start_iteration` method is responsible for doing the iteration, by calling the `iterate` method on the head of the chain, and doing any necessary preparation and cleanup. It implements the `Iterator_Base` methods to terminate the iteration; its `iterate` method calls the final iterated function, and its other `Iterator_Base` methods return straight away. It is therefore responsible for both the head and tail of the chain of iterators, the head being provided by `Iterator_Manager::get_head`, and the tail being its `Iterator_Base` part.

The `Test_File_Generator` class is the client of the framework in this example. It is responsible for implementing the final function that uses the iterated values, and writing the input and output values to file. It writes the input values by calling `write_chained_values` on the iterator returned by the base class `get_head` method.

The `Value_Iterator` is responsible for doing the looping and the recursion. Its `iterate` method runs its loop from `begin` to `end`, relying on its specialisations to provide the next value, given the

```
class Iterator_Base {
 public:
    virtual ~Iterator_Base() {};
    virtual void iterate () = 0;
    virtual int get_expected_count () = 0;
    virtual void write_chained_descriptions (
       FILE* f) = 0;
    virtual void write_chained_values (
       FILE* f) = 0;
    virtual void dispose () = 0;
};
class Value_Iterator : public Iterator_Base
{
 public:
    virtual void iterate ();
    int get_expected_count ();
    void write_chained_descriptions (FILE* f);
    void write_chained_values (FILE* f);
    void dispose ();
 protected:
    Value_Iterator (double* value,
                    double begin,
                    double end,
                    std::string dscr,
                    Iterator_Base* next);
 private:
    virtual double next_value (double v) = 0;
    double begin;
    double end;
    double* value;
    std::string description;
    Iterator_Base* next;
};
class Log_Iterator : public Value_Iterator
{
  public:
    Log_Iterator (double* value,
                  double begin,
                  double end,
                  double step,
                  const std::string& desc,
                  Iterator_Base* next)
    : Value_Iterator (value, begin, end,
                std::string (desc), next),
                step (step)
    {};
private:
    double step;
    double next_value (double v)
    {return v * step;}
};
class Lin_Iterator : public Value_Iterator {
public:
    Lin_Iterator (double* value,
                  double begin,
                  double end,
                  double step,
                  const std::string& desc,
                  Iterator_Base* next)
        : Value_Iterator (value, begin, end,
          std::string (desc), next),
          step (step)
        {};
private:
    double step;
    double next_value (double v)
    {return v + step;}
};
class Iterator_Manager : public Iterator_Base
{
protected:
    Iterator_Manager (Value_Iterator* head);
    ~Iterator_Manager();
    void start_iteration ();
    int get_loop_num() {return loop_num;}
  virtual void iterated_function () = 0;
    Value_Iterator* get_head() {return head;}
private:
    void iterate ();
    void dispose () {;}
    int get_expected_count () {return 1;}
    void write_chained_values (FILE*) {;}
    void write_chained_descriptions (FILE*)
    {;}
    Value_Iterator* head;
    Iterator_Progress* prog;
    int loop_num;
};
```

Listing 4: Listing of the final header file.

```
Value_Iterator::Value_Iterator (
   double* value,
   double begin,
   double end,
   string descr,
Iterator_Base* next) : value (value),
      begin (begin),
      end   (end),
      description (description),
      next  (next)
{}
void Value_Iterator::iterate () {
    for (*value = begin; *value <= end;
*value = next_value (*value)) {
        next->iterate();
    }
}
int Value_Iterator::get_expected_count () {
    int i = 0;
    for (*value = begin; *value <= end;
*value = next_value (*value)) ++i;
    return i * next->get_expected_count();
}
void
Value_Iterator::write_chained_descriptions
(FILE* f) {
    next->write_chained_descriptions(f);
    fprintf (f, ",%s", description.c_str());
}
void Value_Iterator::write_chained_values
(FILE* f) {
    next->write_chained_values(f);
    fprintf (f, ",%0.16e", value);
}
void Value_Iterator::dispose () {
    next->dispose();
    delete this;
}
```

```
Iterator_Manager::Iterator_Manager
(Value_Iterator* head) : prog (0),
          loop_num (0), head (head)
    {}
Iterator_Manager::~Iterator_Manager() {
    delete prog;
    head->dispose();
}
void Iterator_Manager::iterate () {
    ++loop_num;
    prog->update();
    iterated_function();
}
void Iterator_Manager::start_iteration () {
    loop_num = 0;
    prog = new Iterator_Progress (head);
    head->iterate();
    delete prog;
    prog = 0;
}
```

Listing 5: Listing of the final implementation file

current value. This allows specialisations for linear or logarithmic steps, or potentially for some other sequencing method not yet thought of. Within that loop it calls the `iterate` method on its `next` member. Its `write_chained_values` method recurses to the next iterator in the chain, and then writes its own value to the file.

The `Iterator_Progress` class, which is omitted from the listings for brevity, prints a "% complete" message to the console using the expected total number of steps. It calculates the expected number of steps at the start by calling `get_expected_count` on the head iterator, which multiplies each iterator's own number of loops by the next iterator's total number of loops, recursively.

This approach was reasonably satisfactory, but had some drawbacks:

- It was necessary to duplicate the names of the variables: the iterators are named once in the declaration of the client class, and once in the definition of its constructor.

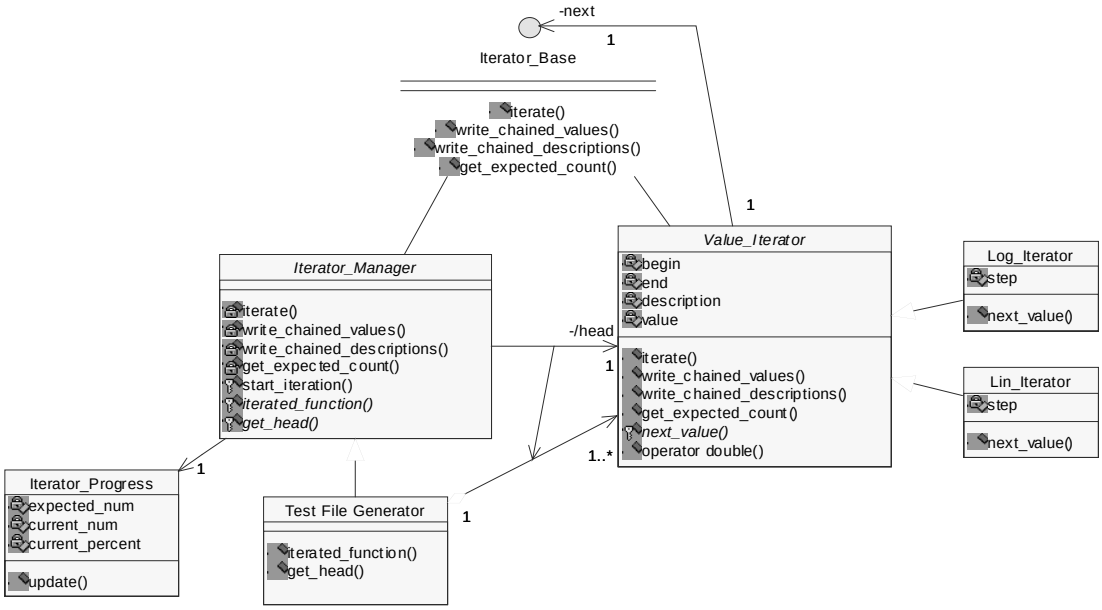
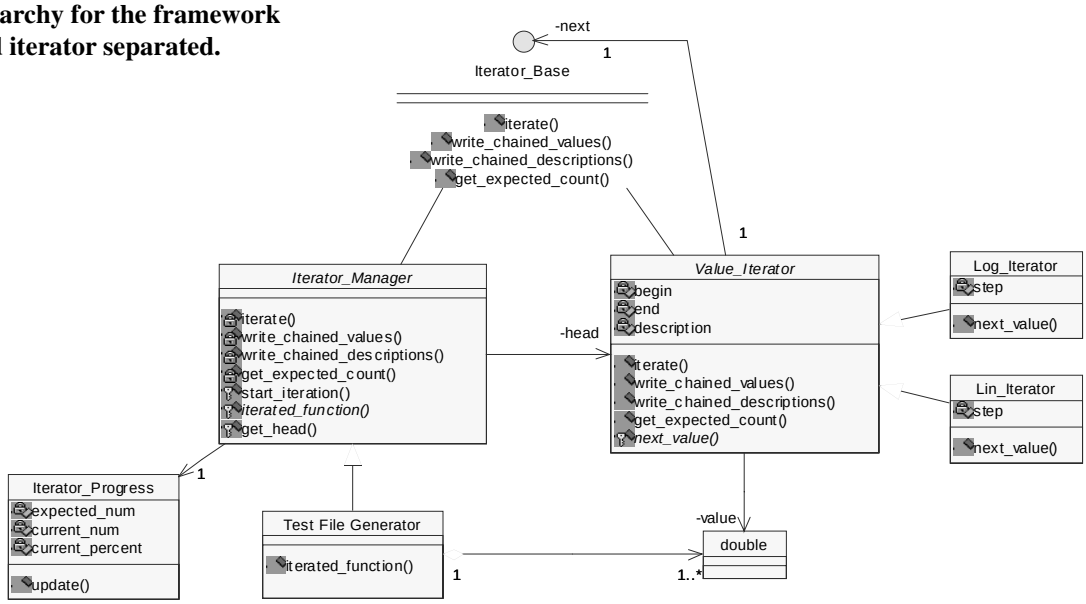


Figure 1: First successful version of the framework

Figure 2: Class hierarchy for the framework with value and iterator separated.



- This decision of linear or logarithmic type of iterator is separated from the definition of its step and limits, although these are clearly related.
- The iterators have to be chained together manually in the constructor, introducing a third place where the iterator is named.

```
class Quadratic_Generator : private
Iterator_Manager {
public:
   Quadratic_Generator () : Iterator_Manager (
    new Lin_Iterator (&a, 0.1, 5.0, 0.1, "a",
    new Lin_Iterator (&b, 10.0, 20.0, 0.1, "b",
    new Lin_Iterator (&c, 0.0, 5.0, 0.1,
    "c", this))))
    {};
    void start ();
private:
    void iterated_function ();
    FILE* f;
    double a, b, c;
};
void Quadratic_Generator::start () {
    f = fopen ("quad.txt", "w");
    get_head()->write_chained_descriptions (f);
    fprintf (f, ",x1,x2\n");
    start_iteration();
    fclose (f);
}
void Quadratic_Generator::iterated_function ()
{
    double z = sqrt (b * b - 4.0 * a * c);
    double x1 = (-b + z) / (2.0 * a);
    double x2 = (-b - z) / (2.0 * a);
    get_head()->write_chained_values (f);
    fprintf (f, ",%.16lf, %.16lf\n", x1, x2);
}
int main (void) {
    Quadratic_Generator().start();
}
```

Listing 6: Listing of example usage for the final version.

Listing 3 shows an example of usage of this approach. The implementation is not shown as it adds little to the discussion.

A Refinement

Most of the drawbacks in the Iterator method derive from the duality of the `Iterator` object, as both the iterator and the value being iterated. This leads to the object being known to both the framework's `Iterator_Manager` class and its derived client class, as shown by the derived association[1] from `Iterator_Manager` to `value_Iterator` in Figure 1. A clearer separation of concerns was found by binding the Iterator objects to members of an enclosing class. This leads to the class model shown in Figure 2. Note that the derived association has been replaced by two distinct associations, each with its own clear purpose.

Listing 4 is the code for the header file of the framework, with the implementation in Listing 5. Assertions, error handling and private/protected constructors/destructors are omitted for brevity. Note the use of the C library FILE I/O mechanism; we found that this was significantly faster than iostream on the implementation used, and performance is fairly important here. A "gold plated" solution might use a policy or helper class to parameterise the choice of output library, but that was not necessary in this project.

Listing 6 gives a code listing for an example usage of the framework. The client creates instances of `Value_Iterator` in its constructor, chaining them together as it does so, and binding them to its own member variables. Thus when the iterate method on the head iterator finally ends up calling the `iterated_function` method on the client, its data members hold the current values for this step and can be used without further ado.

Discussion

By making the iterator objects themselves anonymous, it becomes possible to declare the type of the iterator and initialise the step values on the same line, which makes the connection between log iterator and log steps clearer. It also means that the iterators can be chained by chaining the constructors, rather than by naming them again. In the previous example, adding a new iterator meant

1 In UML, a / in front of a role name or attribute indicates that it can be derived from something else on the model.

[concluded at foot of next page]

With Spirit

by Tim Penhey

Spirit is an object-oriented recursive-descent parser generator framework implemented using template meta-programming techniques. Expression templates allow us to approximate the syntax of Extended Backus-Normal[sic] Form (EBNF) completely in C++. [1]

EBNF is also known as Extended Backus-Naur Form [2]. EBNF is a metasyntax used to formally describe a language. In this example the language is the set of possible expressions that are used to restrict SQL select statements.

The sample code shown is all real code, shown with permission of the owner (a financial institution that wishes to remain anonymous). This piece of code was chosen as a "proof of concept" to show how Spirit works and how it is implemented, both to the management and to the other developers.

The application is a trading system in a bank, and the piece of code is responsible for interpreting what the user enters in a free-text field in the interface used to specify search restrictions. For example, the user may just want to search for certain instruments, or all trades in books starting with the letters B through D. The function `query_parse` (shown below) is the old C version that takes this free text and produces one or more "tokens" for generating the SQL `where` clause.

```
---- some header.h
/*********************************************
 * SQL  Token: consists of :
 * 1. logical operator      : and, or, like
 * 2. mathematical operator : <, >, =, <=,
 *                            >=, <>,
 * 3. value                 : the real value
 *    - i.e. < 30, 30 is the value
 * Before any cell string gets built into an SQL
 * sub-clause, it'll be parsed by query_parse()
 * into a linked-list of SQLTokens, and
 * query_doit() will build using such SQLTokens,
 * instead of cell strings directly.
 *********************************************/

typedef struct _SQLToken
{
  char* logic_op;
  char* math_op;
  char* value;
  struct _SQLToken *next;
} SQLToken;
---- source file.cpp
static SQLToken* query_parse (char *string)
{
  typedef enum { NEUTRAL, LOP, MOP, VALUE }
     STATE;
  char c;
```

[continued from previous page]

having to change the argument to another iterator's constructor, which was a small but noticeable hassle in maintenance.

In the original example progress was indicated by printing a chain of dots, with no indication of how many to expect. When the loops were coded in place in each generator, calculating how many to expect would have required repeating the loop parameters. Extracting the increments into the iterators enables the `get_expected_count` method to do a "dry run", which means a useful progress indicator can be provided without this repetition.

The framework fails to meet the objectives in two points Firstly, the variable name must be duplicated in the constructor of the Iterator object in order to make the binding. While this does seem to be an improvement over repeating the object name twice in the constructor list, it is still less than satisfactory . Secondly, the output column headings written in the generator's start method are still separated from the output of the column data in `iterated_function`. There is therefore a maintenance risk that the column heading could get out of step. No way around this was found that did not result in unjustifiable complexity elsewhere. However, by removing most of the management code, the two relevant lines are now much closer together than in the naïve solution, so the risk is at least to some extent mitigated. An additional minor drawback in this version is that binding the iterator object to a member variable could be seen as breaking the encapsulation of the classes.

Presently the bound member variables must be doubles, which was sufficient for our needs. It would be straightforward to parameterise the Iterator class on the type of bound variable.

Conclusions

The final version meets most of the drivers listed in the introduction:

- It avoids unnecessary complexity and advanced techniques;
- It supports linear and logarithmic steps and progress reporting;
- Most of the duplication is avoided as the iterator objects do not need to be declared in the class.

This framework provides a simple way of producing deep nested loops, with readable client code, clear separation of concerns between client and framework classes, and improved progress reporting. It meets most of the design objectives in a natural, "classic" C++ style, without resort to possibly obscure C++ techniques. This solution is simpler than was expected at the start of the work, which is pleasing.

Peter Hammond
peter.hammond@baesystems.com

References

1 Greg Welch & Gary Bishop, *An Introduction to the Kalman Filter*, TR 95-041, University of North Carolina Chapel Hill, http://www.cs.unc.edu/~welch/kalman/kalmanIntro.html

2 ColdFrame project home page: http://coldframe.sourceforge.net

3 http://libre.act-europe.fr/aunit/main.html

Acknowledgements

The author wishes to thank the reviewers for their helpful comments, particularly Alan Griffiths for the basis for listing 2.

Only a couple of issues ago I was writing about the variety of C++ dialects in use. This article illustrates this: after the author tried using some ideas from "Modern C++ Design" he was inspired to show that "Classic C++" also solves problems. However, some members of the Overload team feel that "Modern C++" offers a better solution.

- ed.

```
  int index = 0, blank = 0;
  SQLToken *token, *tmp=0, *head;
  // fix compiler warning - tmp
  STATE state = NEUTRAL;

  head = sqltoken_alloc();
  token = head;

  while( ( c = string[index] ) &&
     ( c != '\n' ) ) {
    blank = 0;
    switch(state) {
    /*************************************/
    case NEUTRAL:
      switch(c) {
        case ' ':
        case '\t':
          blank = 1;
          ++index;
          break;
        case '+':
        case '|':
          state = LOP;
          break;
        case '<':
        case '>':
        case '=':
        case '!':
        /* if ( first != 0) return NULL; */
        /* only the begin of string may have */
        /* no LOP first = 1; */
          state = MOP;
          break;

        default :
          /* return NULL; */
          state = VALUE;
          break;
      }

  /* alloc space for next SQLToken, if needed */
       if ((token == NULL) && (!blank)) {
         token = sqltoken_alloc();
         tmp->next = token;
       }
       break;

/*************************************/
     case LOP:
       switch(c) {
       case '|':
         while ((c != ' ') && (c!= '\0') &&
            (c != '"') && (c != '>') &&
            (c!= '<') && (c != '=') &&
            ( c != '!'))
           c = string[++index];
         strcat(token->logic_op, "or");
         break;
       case '+':
         while ((c != ' ') && (c!= '\0') &&
            (c != '"') && (c != '>') &&
            (c!= '<') && (c != '=') &&
            ( c != '!'))
           c = string[++index];
         strcat(token->logic_op, "and");
         break;
       default:
         return NULL;
       }

       state = NEUTRAL;
       if ((c != '"') && (c != '>') && (c!= '<')
          && (c != '=') && ( c != '!'))
         index++;
       break;

/*************************************/
     case MOP:
       switch(c) {
       case ' ':
       case '\t':
         state = VALUE;
         index++;
         break;
       case '<':
       case '>':
       case '=':
       case '!':
         strncat(token->math_op, &c, 1);
         index++;
         break;
         default:
         /* if (token->math_op == NULL)
            return NULL; MOP missing */
       state = VALUE;
       }
       break;


/*************************************/
     case VALUE:
       switch(c) {
       case ' ':
         index++;
         break;
       case '"':
          while (((c = string[++index]) != '"')
            && (c != '\0') && (c != '\n'))
           strncat(token->value, &c, 1);
         index++;
         state = NEUTRAL;
         tmp = token;
         token = token->next;
         break;
       default:
         while ((c != ' ') && (c != '\0')&&
            (c != '\n')&& (c != '"'))
           {
            strncat(token->value, &c, 1);
            c = string[++index];
```

```
            }
          state = NEUTRAL;
          tmp = token;
          token = token->next;
        }
        break;
      }
    }
    return head;
}
```

You can see that this code is not very easy to follow, and not overly descriptive in what it does. Clearly it iterates over the character array switching on a remembered state to build up the `SQLToken` instance. However it is not apparent if there is a bug in the code, and should this method need to be extended due to a change in the grammar, much rework may be needed.

A small piece of history. The application was started around 12 years ago and was originally all C. Policy is now that new development should be in C++, updating old code where necessary. So to bring the interface more into line with C++ the signature was changed to:

```
std::vector<SQLToken> query_parse(
   char const* input)
```

The input parameter was not changed to a string as that would not really have gained anything. The calling function had the data as a `char const*`, and that is also the type for the parameter for the parser. Also the `SQLToken` definition changed to use `std::string`:

```
struct SQLToken
{
      std::string logic_op;
      std::string comp_op;
      std::string value;
};
```

In order to move the legacy function to Spirit, the grammar had to be defined. By meticulous iteration of the existing function with sample input, the following grammar was extracted.

```
comp_op ::= '<' | '<=' | '<>' | '>' | '>=' |
   '=' | '!='
logic_op ::= '+' | '|'
value ::= '"' not_quote+ '"' | not_space+
element ::= (logic_op? comp_op? value)+
```

where `not_quote` is any character except the quote character (`"`), and `not_space` is any character except white space (space, tab, or new line).

Now the documentation of the boost website for Spirit gives a great, easy to follow introduction [3]. The management summary equivalent goes something like this:

- a parser is made up from rules
- rules are place holders for expressions
- expressions are either primitives or combinations

Spirit provides classes that define rules and parsers. It also provides a fairly complete set of primitives. The main primitives used for this example are `spirit::str_p` and `spirit::ch_p`. `str_p` matches a string, and `ch_p` matches a single character.

Expressions can be grouped with brackets, alternatives defined by | (bar character), and combined using the `>>` operator. The bar operator is overloaded in Spirit allowing us to not explicitly wrap alternatives in constructor calls. This is a convenience especially when trying to fit examples in a small text area.

The first two grammar components are quite simple. For now just accept that what is being assigned is some form of rule class and the declaration will come later.

```
comp_op = spirit::str_p("<>") | "<=" | "<" |
   ">=" | ">" | "=" | "!=";
logic_op = spirit::ch_p('+') | '|';
```

The quirky parts of this are that the expressions are evaluated in a short circuit manner, so for the comparison operators you need to list the longest first, so `<>` needs to come before `<` otherwise the `<` will be matched for that expression. The Spirit library does provide a way to get around the short circuit nature with a directive. Directives could be thought of as modifiers to an expression. Here use of the `longest_d` directive would suffice, which would give:

```
comp_op = spirit::longest_d[
   spirit::str_p('<') | '<=' | '<>' | '>' |
   '>=' | '=' | '!=' ]
```

However the choice was to go with the simpler definition and a comment.

Now for the value rule. Some of the predefined character parsers were used for this.

`ch_p('"')` matches the quote character, `~ch_p('"')` matches any character except the quote character, and `+(~ch_p('"'))` matches one or more non-quote characters. So the first part of the value is

```
'"' >> (+(~spirit::ch_p('"'))) >> '"'
```

The alternative to a quote enclosed string is a single word, where the contents of the word is anything that isn't whitespace. Spirit provides a `space_p` that matches whitespace characters, so `~space_p` will match non-whitespace characters. To make a word, we use

```
(+(~spirit::space_p))
```

Most of the time when parsing, whitespace is ignored, however in this case whitespace matters. This rule as it stands actually matches the string `"a b c d"` as `"abcd"`. In order to tell the parser that we are concerned about the whitespace, we use the directive `lexeme_d`. The full rule for value is then:

```
value = '"'
      >> (+(~spirit::ch_p('"')))
      >> '"'
      |
spirit::lexeme_d[(+(~spirit::space_p))];
```

The element then is an accumulation of the other rules. `operator!` is used as zero or one, so the element is then

```
element = +(!logic_op >> !comp_op >> value);
```

The complete definition for the grammar object is then:

```
struct query_grammar : public spirit::
   grammar<query_grammar>
{
  template <typename ScannerT>
  struct definition
  {
    definition(query_grammar const& self)
```

```
    {
      // short circuit, so do longer
      // possibilities first
      comp_op = spirit::str_p("<>") | "<="
         | "<" | ">=" | ">" | "=" | "!=";
      logic_op = spirit::ch_p('+') | '|';
      value = '"'
         >> (+(~spirit::ch_p('"')))
         >> '"' | spirit::lexeme_d[
         (+(~spirit::space_p)) ];
       element = +(!logic_op >>
          !comp_op >> value);
       BOOST_SPIRIT_DEBUG_RULE(comp_op);
       BOOST_SPIRIT_DEBUG_RULE(logic_op);
       BOOST_SPIRIT_DEBUG_RULE(value);
       BOOST_SPIRIT_DEBUG_RULE(element);
      }
    spirit::rule<ScannerT> comp_op,
       logic_op, value, element;
    spirit::rule<ScannerT> const& start()
       const { return element; }
  };
};
```

The `BOOST_SPIRIT_DEBUG_RULE` macro enables some very useful debugging output which is handy when tracing your grammar if it is going wrong. A quick interactive test program allows us to test the grammar.

```
int main()
{
   std::cout << "> ";
   std::string input;
   std::getline(std::cin, input);
   query_grammar parser;
   while (input != "quit") {
      if (spirit::parse(input.c_str(), parser,
         spirit::space_p).full)
         std::cout << "parse succeeded";
      else
         std::cout << "parse failed";
      std::cout << "\n> ";
      std::getline(std::cin, input);
   }
}
```

Ths was used to prove that the grammar was correct. The next challenge is how to get the parser to populate the vector of `SQLToken` objects while parsing? I want the `SQLToken` object to be populated during parsing and, once a complete token has been processed `(!logic_op >> !comp_op >> value)`, it should be pushed on to the vector.

The interesting part of handling assignment is that the definition `struct` constructor takes a constant reference to the outer grammar structure, so you cannot change normal member variables. This leaves the choices of mutable and references, and personally I tend to shy away from mutable where there is another choice. So the outer grammar `stuct` holds references to objects that we want to populate.

```
struct query_grammar :
   public spirit::grammar<query_grammar>
{
    // definition structure here...
    query_grammar(std::vector<SQLToken>&
       tokens, SQLToken& token)
       : tokens_(tokens), token_(token) {}
    std::vector<SQLToken>& tokens_;
    SQLToken& token_;
};
```

The next step is to add the actions to the rules, and this is done through the use of "actors". There are a number of predefined actors. The main one used here is `assign_a`. The function call operator on this actor takes one or two parameters. The first parameter is a reference to the string object to populate. If the second parameter is passed in, it assigns the second parameter to the first, and if not, the text that is matched for the rule is assigned.

There is the situation where we want to assign `"and"` when the parser finds `'+'`, and `"or"` for `'|'`, so the `logic_op` rule is changed to look like this:

```
logic_op = spirit::ch_p('+')[spirit::assign_a(
   self.token_.logic_op, "and")]
      | spirit::ch_p('|')[spirit::assign_a(
   self.token_.logic_op, "or")];
```

Since the action is being used on the components of the rule, the definition now has to specify `ch_p('|')` instead of just `'|'`, as there is no `operator[]` on a char.

For the value, if it was quote enclosed, the value is the contents of the string without the quotes, otherwise the value is the whole single word, so the actor is applied to the parts of the value rule, not on the rule as a whole.

```
value = '"'
   >> (+(~spirit::ch_p('"')))
   [spirit::assign_a(self.token_.value)]
   >> '"' | spirit::lexeme_d[
   (+(~spirit::space_p))
   [spirit::assign_a(self.token_.value)] ];
```

The comparison operator can be handled at the whole rule level as the text of the parsed rule is the string value that we want to store for the `SQLToken`. This is achieved by specifying the action for the `comp_op` rule in the element.

```
element = +(!logic_op
   >> !(comp_op[spirit::assign_a(
   self.token_.comp_op)]) >> value);
```

The last part of the parsing is to add the token to the vector. One way of doing this is through a functor object. Standard Spirit functors need to handle two `char const*` parameters. These are the start and end of the "match" for the rule. In this case they aren't used at all, but instead the functor operates on the references that it is constructed with.

```
struct push_token
{
   push_token(std::vector<SQLToken>& tokens,
      SQLToken& token) : tokens_(tokens),
      token_(token) {}
   void operator()(char const*,
      char const*) const
     {
      tokens_.push_back(token_);
      // reset token_ to blanks
      token_ = SQLToken();
     }
   std::vector<SQLToken>& tokens_;
   SQLToken& token_;
};
```

To incorporate this functor into our element rule, we specify it as the action and construct it with the same references as the grammar.

```
element = +(!logic_op
   >> !(comp_op[spirit::assign_a(
   self.token_.comp_op)])
   >> value)[push_token(self.tokens_,
   self.token_)];
```

Now it's done. After testing the results, which to my initial surprise worked perfectly, the old function was replaced with this:

```
namespace {
  using namespace boost;
  struct push_token
  {
    push_token(std::vector<SQLToken>& tokens,
       SQLToken& token) : tokens_(tokens),
       token_(token) {}
    void operator()(char const*,
       char const*) const
    {
      tokens_.push_back(token_);
      // reset token_ to blanks
      token_ = SQLToken();
    }
    std::vector<SQLToken>& tokens_;
    SQLToken& token_;
  };
  struct query_grammar : public spirit
     ::grammar<query_grammar>
  {
    template <typename ScannerT>
    struct definition
    {
      definition(query_grammar const& self)
      {
        // short circuit, so do longer
        // possibilities first
        comp_op = spirit::str_p("<>") | "<=" |
           "<" | ">=" | ">" | "=" | "!=";
        // + -> and, | -> or.  Could now
        // easily add in "and" and "or"
        logic_op = spirit::ch_p('+')[spirit
           ::assign_a(self.token_.logic_op,
           "and")] | spirit::ch_p('|')[spirit
           ::assign_a(self.token_.logic_op,
           "or")];
        // values are single words or
        // enclosed in quotes.
        value = '"' >> (+(~spirit::ch_p('"')))
           [spirit::assign_a(self.token_.value)]
        >> '"' | spirit::lexeme_d[
           (+(~spirit::space_p))
           [spirit::assign_a(self.token_.value)]
           ];
        // EBNF:  (logic_op? comp_op? value)+
        // parsing fails if there are no values.
        element = +(!logic_op
        >> !(comp_op[spirit::assign_a(
           self.token_.comp_op)])
        >> value)[push_token(self.tokens_,
           self.token_)];
      }
      spirit::rule<ScannerT> comp_op, logic_op,
         value, element;
      spirit::rule<ScannerT> const& start() const
         { return element; }
    };
    query_grammar(std::vector<SQLToken>& tokens,
       SQLToken& token)
       : tokens_(tokens), token_(token) {}
       std::vector<SQLToken>& tokens_;
       SQLToken& token_;
    };
    std::vector<SQLToken> query_parse(
       char const* input)
    {
      Logger logger("gds.query.engine.parse");
      GDS_DEBUG_STREAM(logger)
         << "query_parse input: " << input;
        std::vector<SQLToken> tokens;
        SQLToken token;
        query_grammar parser(tokens, token);
        if (spirit::parse(input, parser,
           spirit::space_p).full)
        {
          if (logger.isDebugEnabled()) {
            for (unsigned i = 0;
            i < tokens.size();
            ++i) GDS_DEBUG_STREAM(logger)
            << tokens[i];
          }
        }
        else
        {
          GDS_DEBUG(logger, "parse failed");
          tokens.clear();
        }
      return tokens;
    }
  } // anon namespace
```

An anonymous namespace is used instead of the old static C function, some logging was added using our logging classes, but apart from that, the code went in without other modifications.

In total, I achieved a reduction of about 40 lines of code, which in itself is completely meaningless. The general complexity of the code increased, but at least in my opinion, it is now more maintainable and extensible. Should the client want to make modifications to the grammar it is now a relatively simple operation compared to the nightmare of altering the original embedded switch statements.

Special thanks to Phil Bass and David Carter-Hitchin for reviewing this article.

Tim Penhey
<tim@penhey.net>

References

1 http://www.boost.org/libs/spirit/doc/introduction.html
2 http://en.wikipedia.org/wiki/Extended_Backus-Naur_form
3 http://www.boost.org/libs/spirit/doc/basic_concepts.html