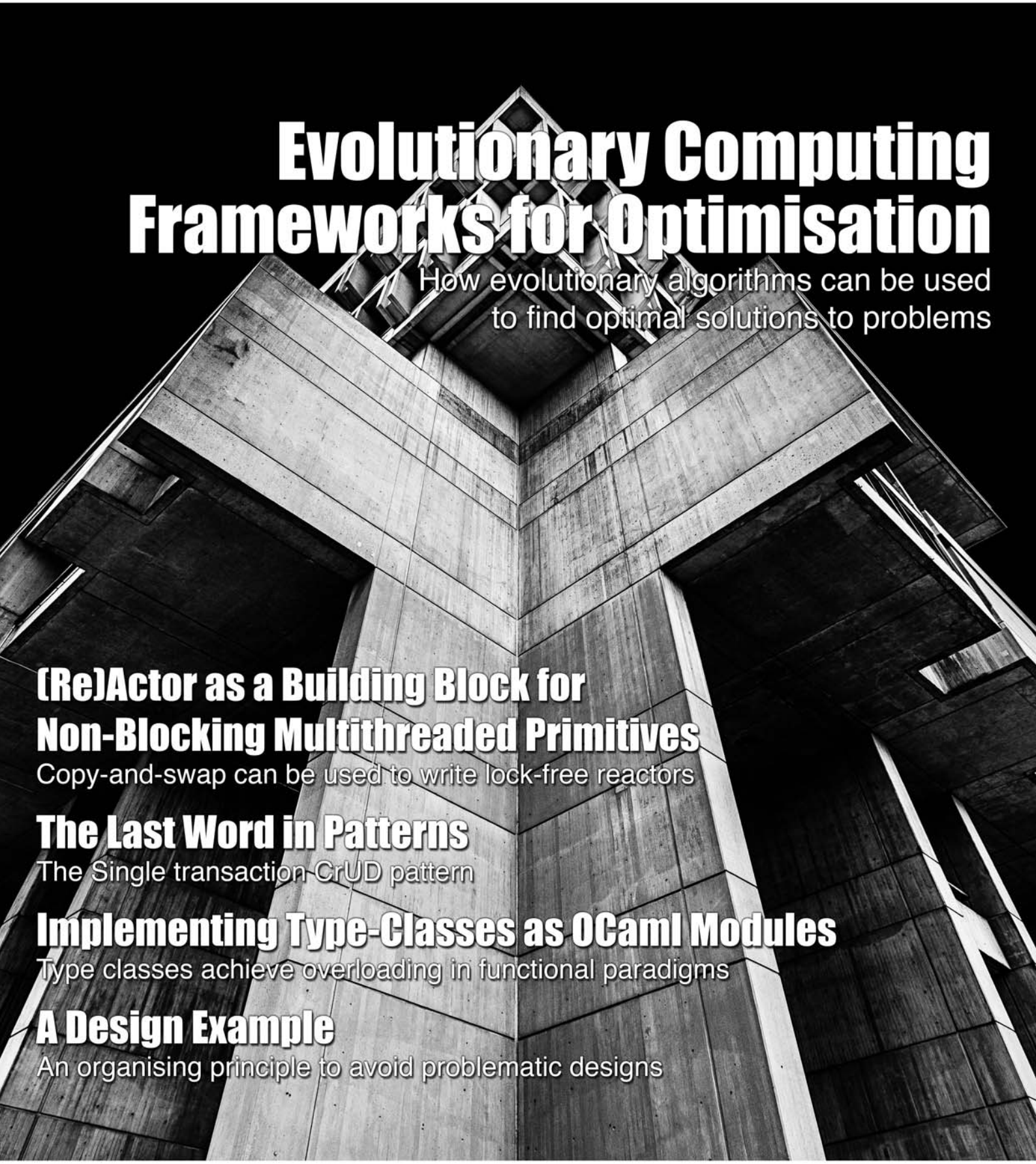
overload 142
DECEMBER 2017 £3



Evolutionary Computing Frameworks for Optimisation

How evolutionary algorithms can be used to find optimal solutions to problems

(Re)Actor as a Building Block for Non-Blocking Multithreaded Primitives

Copy-and-swap can be used to write lock-free reactors

The Last Word in Patterns

The Single transaction CrUD pattern

Implementing Type-Classes as OCaml Modules

Type classes achieve overloading in functional paradigms

A Design Example

An organising principle to avoid problematic designs

A magazine of ACCU
ISSN: 1354-3172

Martin-Baker

SOFTWARE ENGINEER - DENHAM

Martin-Baker is the world's leading manufacturer of aircrew escape and safety systems. It is the only company that can offer a fully integrated escape system that satisfies the very latest in pilot operational capability and safety standards. We are looking for talented and dedicated people that are interested in making a real difference and, like us, are proud to work for a Company whose products have saved over 7550 lives.

We have a fantastic opportunity for Software Engineers working in the Systems Engineering department at the Denham site. As a software engineer you will be joining the team responsible for development and maintenance of critical software assets integral to the growth of Martin-Baker.

This includes:

- Embedded software for safety-critical applications
- Embedded and PC-based test software, used to for verification and in-service usage
- PC-based simulation software, used to model the behaviour of ejection seats during use

Typical work for the Software Engineering Group includes:

- Working with Systems Engineers to specify system and software requirements
- Implementing innovative software solutions
- Delivering a technology refresh to the software toolchain
- Improving Software Group department productivity through targeted infrastructure enhancement projects

We are looking for good engineers who focus on producing high integrity software solutions, work with a wide range of toolsets, and reason about software requirements and design independently of the technology used for implementation.

Martin-Baker offers:

- A competitive salary
- A Non-Contributory Pension Scheme and Life Assurance Scheme
- A Healthcare Scheme
- The equivalent of 25 days annual leave plus statutory holidays
- The opportunity to develop your skills and progress your career
- A great working environment where you will work alongside experts who can help you to obtain a wide understanding of our business

The ideal candidate will be degree-qualified in an appropriate engineering discipline and have previous experience using at least 2 of the following: C (including MISRA an advantage), C++, C#, Java and Ada (including Ada 2012 and SPARK 2014).

For all successful candidates, Martin-Baker will undertake background security checks. As part of this, we will need to confirm your identity, employment history and address history to cover the past five years as well as your nationality, immigration status and criminal record. For positions that require Security Clearance, the successful candidate must hold or be willing to obtain security clearance up to the relevant level for the role.

To apply for this position please email send your covering letter, CV and current salary expectations by email to the Human Resources department at recruitment@martin-baker.co.uk.

Engineering For Life

EJECTION SEAT

Martin-Baker

7553* Aircrew Lives Saved

*October 2017

www.martin-baker.com

OVERLOAD 142

December 2017

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Balog Pal
pasa@lib.h

Ben Curry
b.d.curry@gmail.com

Paul Johnson
paulf.johnson@gmail.com

Klitos Kyriacou
klitos.kyriacou@gmail.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Philipp Schwaha
<philipp@schwaha.net>

Anthony Williams
anthony@justsoftwaresolutions.co.uk

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 143 should be submitted by 1st January 2018 and those for Overload 144 by 1st March 2018.

The ACCU

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

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

Overload is a publication of the ACCU

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

Copyrights and Trade Marks

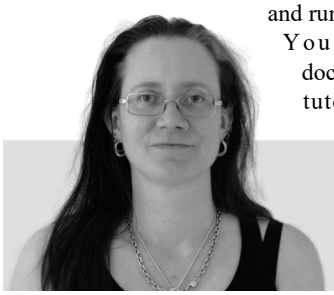
Too Fast! Too slow! Too right!!

Many products over-promise. Frances Buontempo muses on how to get things just right.

I was too slow to start thinking what to write an editorial on in this issue. The last two months have flown by in what felt more like a week, though maybe I wasn't paying full attention. I should try to keep a record of all the ideas I start thinking of writing about. No doubt, trying to decide how to record these and make sure I could understand my notes later when I wanted to use them would give me another excuse for not getting round to writing an editorial. Documentation is a waste of space, right?

Have you ever asked a colleague to demonstrate how to do something, only to watch them typing at 100wpm, using keyboard shortcuts and ended up none the wiser as to what they just did. Or possibly worse, they say, "I just run a script I found on the internet". Your colleague has zoomed through what they do and you have learnt nothing from them. "Too fast!" you mutter under your breath. Perhaps you persuade them to write documentation, which can cause another set of problems we'll come to later. Perhaps they follow the current trend to make a screen capture of what they are up to, making sure they use mouse clicks to show what to do, clearly talking through every single step in full detail. The five minute task now has a 20 minute video. Better than 200 pages of documentation? Or worse? You can copy and paste the commands from the docs but not a screencast. If considerate, they may provide scripts too, but it now takes 20 minutes to watch and you have to make notes on where the pertinent bits are rather than just highlight a couple of lines in a document. The cunning may manage to run the talk at 'chipmunk' speed (extra fast, making the voice high-pitched and squeaky) so it only takes 5 minutes to watch. However you work round this, you have gone from the instructions being way too fast to being way too slow. *Sigh* Perhaps new 'AI' algorithms combined with speech to text APIs let you build a table of contents, or just translate their words back into text you can grep. Over-engineered? Perhaps; there must be a better, quicker way.

Perhaps you are provided with documentation instead. We have moved away from comprehensive write-ups filling two lever arch files or a print out of the source code. You've seen the pictures of Margaret Hamilton with the stack of Apollo Guidance Computer source code, forming a pile of paper as tall as she is [NASA]. Who prints source code out nowadays? I did get embroiled in an attempt to find a bug in some FORTRAN code a couple of years ago, and a younger team member did print out the unstructured 'function' but we couldn't find a corridor long enough for all the paper. He did find the bug. Respect! Nowadays you probably expect a short README file showing how to get up and running quickly when you try new code. You're happy to dig into API documentation, or skim through a wiki or tutorial if needed. Sometimes just a README is enough. Sometimes it isn't. There's always a sweet-spot and it depends on the context. How long does it take a new starter on your team to get up to speed? Can they release new functionality within a week? Does it take them a month to even get a computer in the first place? Or two months for a door pass. (True story. Don't ask.)



Documentation and instructions can be found in various places and formats. You may find an online discussion group for software you are using, or failing that, resort to *StackOverflow*. When do you ask a question? As soon as you are stuck? Do you immediately get yelled at for asking a duplicate question? Or do you spend hours reading what's already been posted? Do you only resort to online help after spending a few weeks trying to solve the problem yourself only to find the person who answers is sitting next to you? More generally, online applications often have a questions section on their website. How many times does some rough and ready AI/machine learning algorithm suggest items from a FAQ section that frankly have nothing to do with your problem, and you are forced to type it in and promised your query will be responded to within 24 hours? On the dot of 24 hours later, you get an automated response assuring you of the importance of your custom and that your problem is being investigated. To be fair, a timely response is good manners. Promising someone you'll just be five minutes and then taking an hour or two is rude. If you say five minutes, get back in five minutes with a status update. Conversely, if someone claims they will be "Just five more minutes," respect that and give them the space they've asked for. Of course, the "just five more minutes" is often a symptom of over-optimistic guessing or a play for extra time. Computers can give over-confident completion times too. You've been there:

- 1 minute to completion, which then goes up to 1 hour
- 0 seconds left, staying in that state for at least five minutes
- Updating Visual Studio – why does it take so long? What is it *doing*??

I presume the scripts know how many bytes or steps there are in total and attempt to report time left by approximating the velocity. It's often clearer if the code reports in units it can measure, avoiding the surprising blur that happens with a conjectured speed or velocity. Using appropriate units can make things less confusing, clearer and even accurate.

Stating the average fuel consumption in miles per gallon (bearing in mind varying definitions of gallons and other units) seems sensible. Does your car have a fuel consumption of 50 miles per gallon? How often do you manage this? Perhaps the units were correct, but the word 'average' needs disclaimers in the small print. What about your CO_2 emissions? Perhaps you have an electric car, so try to calculate a miles per gallon equivalent. How far can you go on a full tank (gas or electric)? Does it matter if you have the lights on?? Does it depend on how fast or slow you

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

drive? Perhaps you don't have a car. You may have used middleware though, or have seen statistics on throughput and latency in a datasheet. Some middleware named "Faster than light" guarantees to deliver one million messages per second under specific conditions; there are always conditions, though messages don't seem to travel faster than light. [Tibco] Jokes aside, having a ball-park maximum can be useful. This allows you to do back of the envelope calculations to see what's possible, and how to carve up your messages and micro-services or other software. Complexity notation also provides a way to analyse the possible speed or memory usage of an algorithm. An amortised or worst-case scenario is as useful as your average fuel consumption. The worst (or best) case may almost never happen, but allows you to compare algorithms or automobiles. It may almost always happen too, if you keep trying to sort already sorted data with a naive implementation of the Quicksort algorithm. Either way, if your code is too slow, you may need to start profiling to see what's really going on, but big-O notation gave you a starting point.

Many products give upper bounds on performance. Consider a dandruff shampoo that removes "up to 100% of flakes". I, for one, am relieved to know it won't give 110%, presumably dissolving my skull in the process. Some maximum limits are enforced by science – as we know the speed of light, again under specific conditions, is a hard limit. 100% is often the limit on how much you can remove or lose, unless you trade a contract for difference (CFD) – in which case losses can exceed original investments. One product that almost never gives 100% is broadband, at least in the UK, or certain parts of the UK. How fast is your broadband? Or how slow? What were you promised? An 'up to' I presume. Make sure you write in and complain if it ever exceeds this!

I recently heard about a new attempt to break the land-speed record aiming for 1,000 miles per hour. [Bloodhounds] Managing to go over 100 miles per hour back in 1904 [Redbull] was significant. Wikipedia [Wikipedia_1] tells me the 117 or so miles of a full orbit of M25 (London Orbital) has been made in under an hour, late at night when no law enforcement officers were around. The M25 is often more like a car park than a motorway. In fact, signs often suggest a speed of (up to) 40 miles per hour due to congestion. Usually no one does what they are told, just pootling along at 1 mile per hour. Somebody, somewhere needs to invent a Star-Trek style transporter, quickly. Beside the saving of time and the lack of pollution, "Transporting really is the safest way to travel," at least according to Geordi La Forge [MemoryAlpha]. You may travel faster than the speed of light, though might miss your target destination by up to (there it goes again) 4 metres.

Back to reality. Do you need to deal with large data sets on a regular basis? Or grep giant log files? As you build up a script to find needles in haystacks, you probably try them out on *small* data first, to verify they do what you want. You probably build up regex to hunt in log files gradually, checking it does match some examples and furthermore doesn't match other close but incorrect examples. Under stress, this sounds like it will slow you down, though the temptation to try it on all your data and announce, "There are no matches" can take you more time in the long run. "More haste, less speed" as an old saying goes. Going too quickly can have the overall effect of slowing you down. If you do try your scripts and programs on small data sets first, where do you get that data? Some people are horrified at the thought of using artificial datasets. However, you can create artificial data to cover all the edge cases and combinations, with one or two examples of each. Real data may not provide the perfect storm you need to test, so make some up. Someone somewhere will tell you this is a waste of time. I disagree. Real data may uncover other problems – partially filled or invalid fields, fragmented records, formats that don't match the thousand page document you read and so on. However, a one in a million event will break your system nine times out of ten (to misquote Terry Pratchett), so try out the black swan events in a test setup. Test your code on small sample data too, but you need both angles to be better covered.

To build up your understanding of a problem domain or technology, you probably try a small experiment. Baby steps first. You might even build an end to end system, and pay attention to the logging and data feeds, just using static data. Someone might complain that you can't test end to end until you have a live data feed. Prove them wrong! When you set up your logging, watch out for too much noise. If you cry "Warning!" over and over will anyone take any notice? Does your log file have known, expected and therefore ignored 'ERROR' lines over and over, slowing down your search for something specific? Do they get archived away too quickly for you to even search in the first place?

Some things are too fast. Some things are too slow. Some things, once in a while, are just right. How can you achieve this Goldilocks sweet-spot? The planet Earth is in the so-called Goldilocks zone – at just the right distance from our Sun to support life. This circumstellar habitable zone [Wikipedia_2] has a long history, and gets refined over time. Starting with supporting liquid water, we've added atmosphere requirements and this will probably continue to change. And yet, of several million planets in goldilocks zones, we've only found one with life on so far. Are you doing things just right, to allow life, ideas and creativity to flourish? I'm not suggesting that you need to do something earth-moving to hit a sweet-spot, but the analogy with a code-base, project plan or team being habitable is a recurring theme. Neither too hot, too cold, but just right.

References

[Bloodhounds] http://www.bloodhoundssc.com/

[MemoryAlpha] http://memory-alpha.wikia.com/wiki/Transporter

[NASA] https://www.nasa.gov/feature/margaret-hamilton-apollo-software-engineer-awarded-presidential-medal-of-freedom

[Redbull] https://www.redbull.com/gb-en/history-of-land-speed-record-cars

[Tibco] https://www.tibco.com/sites/tibco/files/resources/ds-ftl.pdf

[Wikipedia_1] https://en.wikipedia.org/wiki/M25_motorway#Racing

[Wikipedia_2] https://en.wikipedia.org/wiki/Circumstellar_habitable_zone

CAS (Re)Actor for Non-Blocking Multithreaded Primitives

Lock free programming can be difficult. Sergey Ignatchenko shows how copy and swap can work for reactors.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Hare, and do not necessarily coincide with the opinions of the translators and *Overload* editors; also, please keep in mind that translation difficulties from Lapine (like those described in [Loganberry04) might have prevented an exact translation. In addition, the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

Those of you who happen to follow my ramblings, probably already know that I am a big fan of so-called (Re)Actors (see, for example, [NoBugs10], [NoBugs15], and [NoBugs17]).

Very, very briefly, a (Re)Actor is a thing which is known under a dozen different names, including 'Actor', 'Reactor', 'event-driven program', and 'ad hoc state machine'. What is most important for us now is that the logic within a (Re)Actor is inherently thread-agnostic; in other words, logic within the (Re)Actor runs without the need to know about inter-thread synchronization (just as a single-threaded program would do). This has numerous benefits: it simplifies development a lot, makes the logic deterministic and therefore testable (and determinism further enables such goodies as post-mortem production debugging and replay-based regression testing), tends to beat mutex-based multithreaded programs performance-wise, etc. etc. And in 2017, I started to feel that the Dark Ages of mutex-based thread sync were over, and that more and more opinion leaders were starting to advocate message-passing approaches in general (see, for example, [Henney17] and [Kaiser17]) and (Re)Actors in particular.

Next, let's note that in spite of the aforementioned single-threaded (or, more precisely, thread-agnostic) nature of each single (Re)Actor, multiple (Re)Actors can be used to build Shared-Nothing near-perfectly-scalable multi-threaded/multi-core systems [NoBugs17]. This observation has recently led me to a not-so-trivial realization that in quite a few cases, we can use (Re)Actors to… <drumroll /> implement non-blocking multithreaded primitives. The specific problem I was thinking about at that point, was a multiple-writer single-reader (MWSR) blocking-only-when-necessary queue with flow control, but I am certain the concept is applicable to a wide array of multithreaded primitives.

Basic Idea – CAS (Re)Actor

As noted above, distributed systems consisting of multiple (Re)Actors are known to work pretty well. Basically, in what I call a (Re)Actor-fest architecture, all we have is a bunch of (Re)Actors, which exchange messages with each other, with nothing more than this bunch of (Re)Actors in sight. Apparently, this model is sufficient to implement any distributed real-world system I know about (and very practically too).

Sergey Ignatchenko has 20+ years of industry experience, including being an architect of a stock exchange, and the sole architect of a game with hundreds of thousands of simultaneous players. He currently writes for a software blog (http://ithare.com), and translates from the Lapine language a 9-volume book series 'Development and Deployment of Multiplayer Online Games'. Sergey can be contacted at sergey.ignatchenko@ithare.com

Compare-and-Swap

In computer science, compare-and-swap (CAS) is an atomic instruction used in multithreading to achieve synchronization. It compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value. This is done as a single atomic operation. [Wikipedia-1]

Now, let's try to use pretty much the same idea to build a multithreaded primitive (using (Re)Actors with an ultra-small state). Let's start with the following few basic concepts:

- We have one or more (Re)Actors

  Each of these (Re)Actors has its state fitting into one CAS block (i.e. the whole state be processed within one CAS operation). Let's call these (Re)Actors 'CAS (Re)Actors'.
- When we're saving the state to the CAS block, all kinds of compression are permissible, as long as we guarantee that the state always fits into one single CAS block. In particular, all kinds of bitfields are perfectly fine.
- All interactions between (Re)Actors are implemented as message exchanges (i.e. no (Re)Actor can access another (Re)Actor's state, except via sending a message asking to perform a certain operation).

  As for the nature of messages – it depends, and in theory they can be as complicated as we wish, but in practice most of the time they will be as simple as a tuple (enum message_type, some_int_t parameter)

As soon as this is in place, we can write and use our (Re)Actors as shown in Listing 1, annotated with (a), (b), (c) and (d) to correspond with the following explanation. The logic within the infinite **`while`** loop with **`compare_exchange_weak`** inside is very standard for CAS-based primitives. First, we're reading the data (in our case, we're doing it in constructor). Then, we're going into an infinite loop: (a) calculating a new value for the CAS block; (b) executing `compare_exchange_weak()`. If `compare_exchange_weak()` returns true (c), our job is done, and we can return the value. If, however, **`compare_exchange_weak()`** returns false, this guarantees that the CAS block wasn't changed, so we can easily discard all our on-stack changes to bring the system to the exact state which was before we started (but with an up-to-date value for **`last_data`**), and try again (d). In practice, it is extremely rare to have more than 2–3 rounds within this 'infinite' loop, but in theory on a highly contentious CAS block, any number of iterations is possible.

Another way to see it is to say that what we're doing here is an incarnation of the good old optimistic locking: we're just trying to perform a kinda-'transaction' over our CAS block, with the kinda-'transaction' being a read-modify-write performed in an optimistic manner. If a mid-air collision (= "somebody has already modified the CAS block while we were working") happens, it will be detected by

```
using CAS=std::atomic<CAS_block>;
CAS global_cas;//accessible from multiple threads
               //in practice, shouldn't be global
               //but for the example it will do
class ReactorAData { //state of our ReactorA
  CAS_block data;

  public:
  ReactorAData() { ... }

  private:
  int OnEventX_mt_agnostic(int param) {
    //modifies our data
    //absolutely NO worries about multithreading
    //here(!) MUST NOT have any side effects
    //such as modifying globals etc.
    //...
  }
  //other OnEvent*_mt_agnostic() handlers
  friend class ReactorAHandle;
};

class ReactorAHandle {//'handle' to the state of
                      // ReactorA
  CAS* cas; //points to global_cas
  ReactorAData last_read;

  public:
  ReactorAHandle(CAS* cas_) {
    cas = cas_;
    last_read = cas->load();
  }
  int OnEventX(int param) {
    while(true) {
      ReactorAData new_data = last_read;
      int ret =
       new_data.OnEventX_mt_agnostic(param);//(a)
      bool ok = cas->compare_exchange_weak(
            last_read.data, new_data.data );//(b)
      if( ok )
        return ret;//(c)
      //(d)
    }
  }
  //other OnEvent*() handlers
};
```

Listing 1

compare_exchange_weak(), and – just as for any other optimistic locking – we just have to rollback our kinda-'transaction' and start over.

That's pretty much it! We've got our multithread-safe event-handling function **OnEventX()** for **ReactorAHandle**, while our **OnEventX_mt_agnostic()** function is, well, multithread-agnostic. This means that we do NOT need to think about multithreading while writing **OnEventX_mt_agnostic()**. This alone counts as a Big Fat Improvement™ when designing correct multithreaded primitives/algorithms.

Moreover, with these mechanics in place, we can build our multithreaded primitives out of multiple (Re)Actors using the very-simple-to-follow logic of "hey, to do this operation, I – as a (Re)ActorA – have to change my own state and to send such-and-such message to another (Re)ActorB". This effectively introduces a layer of abstraction, which tends to provide a much more manageable approach to designing multithreaded primitives/algorithms than designing them right on top of CAS (which are rather difficult to grasp, and happen to be even more difficult to get right).

Of course, as always, it is not a really a silver bullet, and there are certain caveats. In particular, two things are going to cause us trouble on the way: these are (a) a limitation on CAS block size, and (b) an ABA problem.

On CAS block size

One thing which traditionally plagues writers of multithreaded primitives is a limitation on the CAS block size. Fortunately, all modern x64 CPUs support CMPXCHG16B operations, which means that we're speaking about 128-bit CAS blocks for our (Re)Actors. This, while not being much, happens to be not too shabby for the purposes of our extremely limited (Re)Actors.

To further help with the limitations, we can observe that (rather unusually for CAS-based stuff) we can use all kinds of bit-packing techniques within our **CAS_block**. In other words, if we have to have a field within **ReactorAData::data**, we can use as many bits as we need, and don't need to care about alignments, byte boundaries, etc. In addition, we can (and often should) use indexes instead of pointers (which usually helps to save quite a few bits), etc. etc.

Solving the ABA problem

Another issue which almost universally rears its ugly head when speaking about not-so-trivial uses of CAS is the so-called ABA problem. Very, very roughly it is about the system being in exactly the same state under CAS, while being in a different semantic state (for examples of ABA in action, see, for example, [Wikipedia-2]).

Of course, the same problem would apply to our CAS (Re)Actors too. However, apparently there is a neat workaround. If within our **ReactorAData::data**, we keep a special **ABAcounter** field as a part of our **ReactorAData::data** that is a counter of successful modifications of **ReactorAData::data** (i.e. we'll increment this counter on each and every modification of the **ReactorAData::data**) then we're guaranteed to avoid the ABA problem as long as the **ABAcounter** doesn't overflow. This stands merely because for each modification we'll get a different value of the CAS block, and therefore won't run into 'having the same state' situation, ever.

Now, let's take a look at the question of workarounds for the counter. Let's consider a system with the CPU clock running at 3GHz, and a maximum lifetime of the running program being 10 years. Let's also assume that CAS takes no less than 1 cycle (in practice, it takes 10+ at least for x64, but we're being conservative here). Then, the most CAS operations we can possibly make within one single program run, is 1 CAS/cycle * 3e9 cycles/sec * 86400 sec/day * 365 days/year * 10 years ~= 1e18 CAS operations. And as 1e18 can be represented with mere 60 bits, this means that

> *by using a 60-bit ABA counter, we're protected from ABA even under extremely conservative assumptions.*

NB: 40–48 bit counters will be more than enough for most of practical purposes – but even a 60-bit counter is not too bad, especially as our whole allocation, as discussed above, is 128 bits (at least for x64).

Relaxing the requirement for ABAcounter modifications

As discussed above (with sufficient sizes of **ABACounter**) we can guarantee that no ABA problem occurs as long as we increment **ABAcounter** on each and every modification of our **ReactorAData::data**. However, there are cases when we can provide the same guarantees even when we skip incrementing on some of the modifications. More specifically, we can go along the following lines:

- We divide fields within **ReactorAData::data** into two categories: (a) those fields 'protected' by **ABAcounter**, and (b) those fields 'unprotected' by **ABAcounter**
- Then, we still increment **ABAcounter** on any modification to 'protected' fields, but are not required to increment **ABAcounter** on those modifications touching only 'unprotected' fields
- Then, we're still providing 'no-ABA-problem' guarantees as long as all our 'unprotected' fields have the property that the same value

of those 'unprotected' fields is guaranteed to have the same semantic meaning.

- For example, if we have a 'number of current locks' field within our `ReactorAData::data` – for most of the typical usage patterns, we don't really care why this field got this value, but care only about its current value; this means that whatever we're doing with this field, it is ABA-free even without the `ABAcounter`, so it can be left 'unprotected'.

Conclusions and Ongoing Work

We presented a hopefully novel way for building of non-blocking multithreaded primitives and algorithms, based on 'CAS (Re)Actors' (essentially – (Re)Actors with the size fitting into one CAS block).

This approach is practically interesting because it provides an additional layer of abstraction, and – as a result – allows us to reason about multithreaded primitives/algorithms in terms which don't involve multithreading (in particular, such issues as the semantics of CAS and the ABA problem are out of the picture completely). Instead, the reasoning can be done in terms of distributed systems (more specifically – in terms of Actors, Reactors, event-driven programs, or ad hoc finite state machines). This, in turn, is expected to enable composing of more complicated primitives/algorithms than it is currently possible. In particular, the author is currently working on a MWSR queue with locking-only-when-necessary and providing different means of flow control; when the work is completed he hopes to present that in *Overload* too. ■

References

[Henney17] Kevlin Henney, Thinking Outside the Synchronisation Quadrant, ACCU2017

[Kaiser17] Hartmut Kaiser, The Asynchronous C++ Parallel Programming Model, *CPPCON2017*

[Loganberry04] David 'Loganberry', Frithaes! – an Introduction to Colloquial Lapine!, http://bitsnbobstones.watershipdown.org/lapine/overview.html

[NoBugs10] 'No Bugs' Hare, Single-Threading: Back to the Future?, *Overload* #97, 2010

[NoBugs15] 'No Bugs' Hare, Client-Side. On Debugging Distributed Systems, Deterministic Logic, and Finite State Machines, http://ithare.com/chapter-vc-modular-architecture-client-side-on-debugging-distributed-systems-deterministic-logic-and-finite-state-machines/

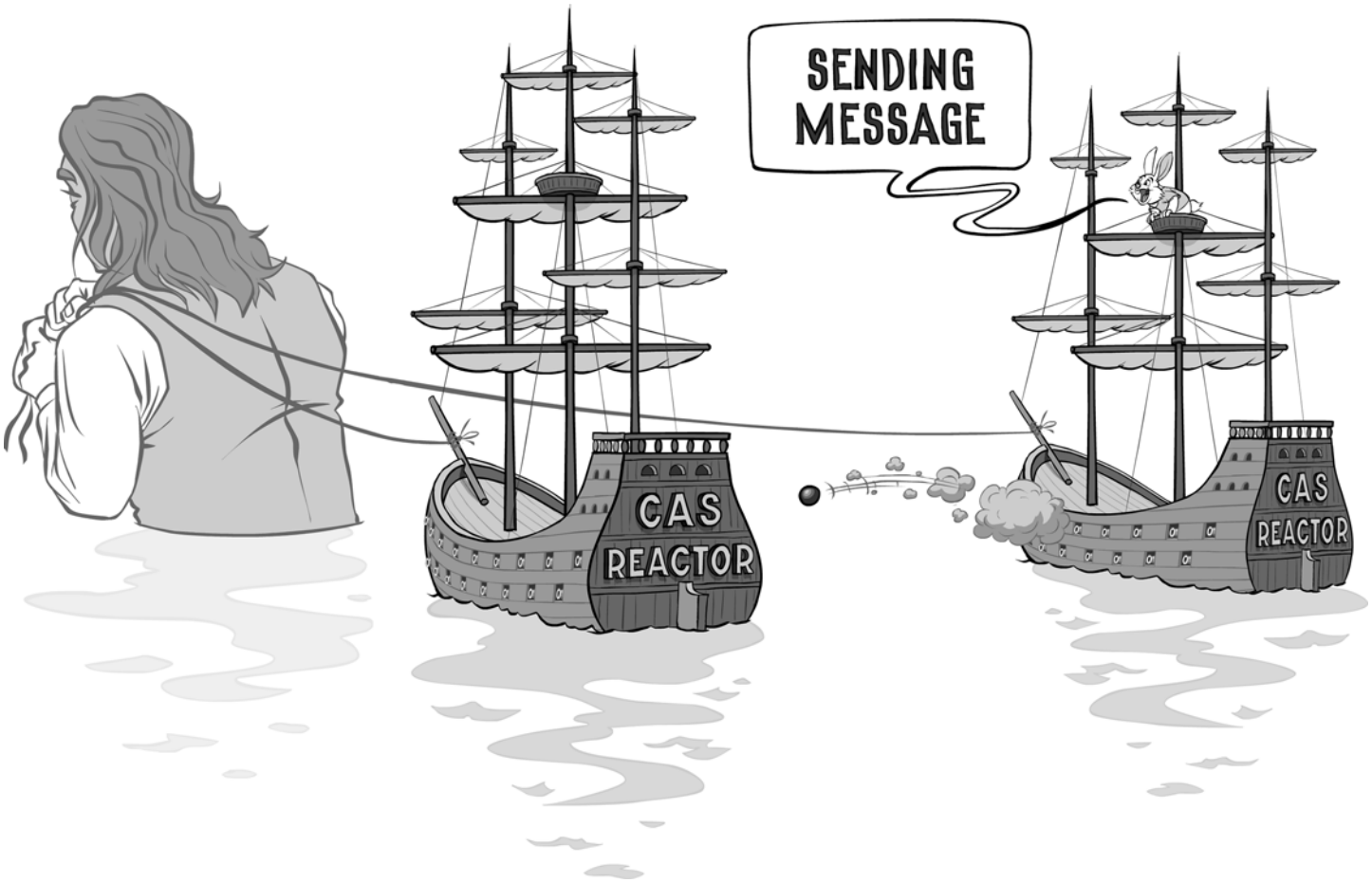
[NoBugs17] 'No Bugs' Hare, Development and Deployment of Multiplayer Online Games, Vol. II.

[Wikipedia-1] Wikipedia, Compare-and-Swap, https://en.wikipedia.org/wiki/Compare-and-swap

[Wikipedia-2] Wikipedia, ABA problem. https://en.wikipedia.org/wiki/ABA_problem

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague



A Design Example

Design issues cause problems. Charles Tolman considers an organising principle to get to the heart of the matter.

I want to underpin the philosophical aspect of this discussion by using an example software architecture and considering some design problems that I have experienced with multi-threaded video player pipelines. The issues I highlight could apply to many video player designs.

Figure 1 is a highly simplified top-level schematic, the original being just an A4 pdf captured from a whiteboard, a tool that I find much better for working on designs than using any computer-based UML drawing tool. The gross motor movement of hand drawing 'in the large' seems to help the thinking process.

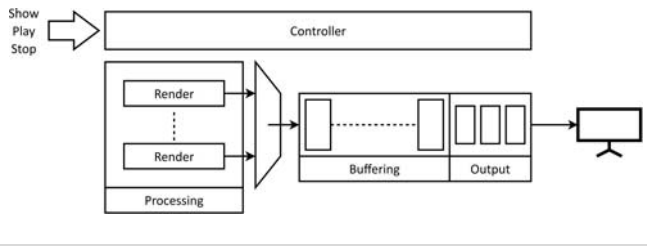


Figure 1

There are 3 basic usual commands for controlling any video player that has random access along a video timeline:

- Show a frame
- Play
- Stop

In this example, there is a main controller thread that handles the commands and controlling the whole pipeline. I am going to conveniently ignore the hard problem of actually reading anything off a disk fast enough to keep a high resolution high frame-rate player fed with data.

The first operation for the pipeline to do is to render the display frames in a parallel manner. The results of these parallel operations, since they will likely be produced out of order, need to be made into an ordered image stream that can then be buffered ahead to cope with any operating system latencies. The buffered images are transferred into an output video card, which has only a relatively small amount of video frame storage. This of course needs to be modeled in the software so that (a) you know when the card is full; and (b) you know when to switch the right frame to the output without producing nasty image tearing artefacts.

These are all standard elements you will get with many video player designs, but I want to highlight three design issues that I experienced in order to get an understanding of what I will later term an 'Organising Principle'.

First there was slow operation resulting in non real-time playout. Second, occasionally you would get hanging playout or stuttering frames. Third, you could very occasionally get frame jitter on stopping.

Slow operation

Given what I said about Goethe and his concept of Delicate Empiricism, the very first thing to do was to reproduce the problem and collect data, i.e. measure the phenomenon WITHOUT jumping to conclusions. In this case it required the development of logging instrumentation software within the system – implemented in a way that did not disturb the real-time operation.

With this problem I initially found that the image processing threads were taking too long, though the processes were doing their job in time once they had their data. So it was slowing down *before* they could get to start their processing.

The processing relied on fairly large processing control structures that were built from some controlling metadata. This build process could take some time so these structures were cached with their access keyed by that metadata, which was a much smaller structure. Accessing this cache would occasionally take a long time and would give slow operation, seemingly of the image processing threads. This cache had only one mutex in its original design and this mutex was locked both for accessing the cache key *and* for building the data structure item. Thus when thread A was reading the cache to get at an already built data item, it would occasionally block behind thread B which was building a new data item. The single mutex was getting locked for too long while thread B built the new item and put it into the cache.

So now I knew exactly where the problem was. Notice the difference between the original assumption of the problem being with the image processing, rather than with the cache access.

It would have been all too easy to jump to an erroneous conclusion, especially prevalent in the Journeyman phase, and change what was *thought* to be the problem. Although such a change would not actually fix the real issue, it could have changed the behaviour and timing so that the problem may not present itself, thus looking like it was fixed. It would likely resurface months later – a costly and damaging process for any business.

In this case the solution here was to have finer grained mutexes: one for the key access into the cache and a separate one for accessing the data item. On first access the data item would be lazily built and thus needed a second mutex to protect the write(build) before read access.

Hanging playout or stuttering frames

The second bug was that the playout would either hang or stutter. This is a good example because it illustrates a principle that we need to learn when dealing with any streamed playout system.

Charles Tolman earned a degree in Electronic Engineering in the 70s, and then moved into software; progressing through assembler to Pascal, Eiffel and eventually C++. He's now involved in large scale C++ development in the CAE domain. Having seen many silver bullets come and go, his interest is in a wider vision of programmer development that encompasses more than purely technical competence. You can contact him at ct@acm.org

the streaming at the output end of the pipeline was happening out of order, a bad fault for a video playout design

The measurement technique in this case was extremely 'old school', simply printing data to a log output file. Of course only a few characters were output per frame, because at 60fps (a typical modern frame-rate) you will only have 16ms per frame.

In this case the streaming at the output end of the pipeline was happening out of order, a bad fault for a video playout design. Depending upon how the implementation was done, it would either cause the whole player to hang or produce a stuttered playout. Finding the cause of this took a lot of analysis of the output logs and many changes to what was being logged. An example of needing to be clear about the limits of one's knowledge and of properly identifying the data that next needed to be collected.

I found that there was an extra 'hidden' thread added within the output card handling layer in order to pass off some other output processing that was required. However it turned out that there was no enforcement of frame streaming order. This meant that the (relatively) small amount of memory in the output card would get fully allocated and this would give rise to a gap in the output frame ordering. The output control stage was unable to fill the gap in the frame sequence with the correct frame, because there was no room in the output card for that frame. This would usually result in the playout hanging.

This is why with a streaming pipeline, where you always have limited resources at some level, allocation of those resources *must* be done in streaming order. This is a dynamic principle that can take a lot of hard won experience to learn.

The usual Journeyman approach to such a problem is just to add more memory, i.e. more resource! This will hide the problem because though processing is still done out of order, the spare capacity has been increased and it will not go wrong until you next modify the system to use more resource. At this point the following statement is usually made:

> But this has been working ok for years!

The instructions I need to tell less experienced programmers when trying to debug such problems will usually include the following:

- Do NOT change any of the existing functionality.
- Disturb the system as little as possible.
- Keep the bug reproducible so you can measure what is happening.

Then you will truly know when you have fixed the fault.

Frame jitter on stop

The third fault case was an issue of frame jitter when stopping playout. The problem was that although the various buffers would get cleared, there could still be some frames 'in flight' in the handover threads. This is a classic multi-threading problem and one that needs careful thought.

In this case when it came time to show the frame at the current position, an existing playout had to be stopped and the correct frame would need to be processed for output. This correct frame for the current position would make its way through to the end of the pipeline, but could get queued behind a remnant frame from the original stopped playout. This remnant frame would most likely have been ahead of the stop position because of the pre-buffering that needed to take place. Then when it came time to re-enable the output frame viewing in order to show the correct frame, both frames would get displayed, with the playout remnant one being shown first. This manifested on the output as a frame jitter.

One likely fix of an inexperienced programmer would be to make the system sit around waiting for seconds while the buffers were cleared and possibly cleared again, just in case! (The truly awful 'sleep' fix.) This is one of those cases where, again due to lack of deep analysis, a defensive programming strategy is used to try and force a fix of what initially seems to be the problem. Again, it is quite likely that this may *seem* to fix the problem, and will probably happen if the developer is under heavy time pressure, but this would be an example where the best practice of taking time to be properly understand the failure mode in a Delicately Empirical way would be compromised by a rush to a solution.

The final solution to this particular problem was to use the concept of uniquely identified commands, i.e. 'command ids'. Thus each command from the controlling thread, whether it was a play request or a show frame request, would get a unique id. This id was then tagged on to each frame as it was passed through the pipeline. By using a low-level globally accessible (within the player) 'valid command id set' the various parts of the pipeline could decide, by looking at the tagged command id, if they had a valid frame that could be allowed through or quietly ignored.

When stopping the playout all that had to be done was to clear the buffers, remove the relevant id from the 'valid command id set' and this would allow any pesky remaining 'in flight' frames to be ignored since they had an invalid command id. This changed the stop behaviour from being an occasional, yet persistent bug, into a completely reliable operation and without the need to use 'sleep' calls anywhere.

Hidden organising principles

In conclusion the above issues dealt with the following design ideas:

- Separating Mutex Concerns.
- Sequential Resource Allocation.
- Global Command Identification.

But I want to characterize these differently because the names sound a little like pattern titles. Although as a software community we have had success using the idea of patterns I think the concept has become rather more fixed than Christopher Alexander may have intended. Thus I will rename the solutions as follows in order to expressly highlight their dynamic behavioural aspect:

- Access Separation.
- Sequential Allocation.
- Operation Filtering.

You might have noticed in the third example the original concept of 'Global Command Identification' represents just one possible way to implement the dynamic issue of filtering operations. Something it has in common with much of the published design pattern work where specific

Being able to perceive and 'livingly think' these mobile thought structures is what we need to do as we make our way to becoming accomplished programmers

example solutions are mentioned. To me design patterns represent a more fixed idea that is closer to the actual implementation.

But there is a difference between the architecture of buildings – where design patterns originated – and the architecture of software. Although both deal with the design of fixed constructs, whether it be a building or code, the programmer has to worry far more about the dynamic behaviour of the fixed construct (their code). Yes – a building architect does have to worry about the dynamic behaviour of people inhabiting their design, but software is an innately active artefact.

Though others may come up with a better naming for the behavioural aspects of the ideas, I am trying to use a more mobile and dynamic definition of the solutions. Looking at the issues in this light starts to get to the core of why it is so hard to develop an architectural awareness. We have to deal with a far more, in philosophical terms, 'phenomenological' set of thoughts. This is why the historical philosophical context can enlighten the problematic aspects of our programming careers.

We need to move our thinking forward and understand the 'Organising Principles' that live *behind* the final design solutions, i.e. how the active processes need to run. Being able to perceive and 'livingly think' these mobile thought structures is what we need to do as we make our way to becoming accomplished programmers.

A truly understood concept of the mobile Organising Principle does not really represent design patterns – not in the way we have them at the moment. It is NOT a static thing. It cannot be written down on a piece of paper. Although it informs any software implementation it cannot be put into the code. If you fix it: You Haven't Got It. Remember that phrase because truly getting and understanding it is the real challenge. The Organising Principle lives *behind* the parts of any concrete implementation.

Despite the slippery nature of the Organising Principle, I shall attempt to explore this more in a subsequent article and give some ideas about what we can do as programmers to improve such mobile perception faculties. ■

The Last Word in Patterns

What can you do in a single transaction in a database? Paul Grenyer writes us his Single CrUD pattern.

Software patterns have their roots in architecture. In 1978, Christopher Alexander published a book called *A Pattern Language: Towns, Buildings, Construction* about the patterns he'd discovered designing buildings [Alexander78]. A pattern can be thought of as a tried and tested way of doing something which can be applied in different contexts. Think about how the OBSERVER or VISITOR pattern is implemented across languages such as Java, Ruby and JavaScript, where the different language idioms dictate slightly different implementations of the same basic pattern.

Software patterns became popular with the publishing of the Gang of Four book, *Design patterns: elements of reusable object-oriented software* [GoF94]. It contains a number of patterns, most of which every developer should know, even if it's to know to avoid the likes of SINGLETON. However, these aren't the only patterns! Indeed, patterns are not created, they are discovered and documented. Whole conferences [Europlop] are dedicated to software patterns, where delegates are encouraged to bring their pattern write-ups for appraisal by their peers and the experts.

When I joined ACCU in 2000, I was encouraged by another member to write for the group's magazine, but I didn't think I'd have anything to contribute that someone hadn't already thought of and written about. As I gained experience, I found I had quite a lot to write about and to challenge.

In the same way, you'd have thought that 23 years after the Gang of Four book most, if not all, of the software patterns had been discovered and documented. Of course, they haven't and I believed, from checking with industry experts, that what I'm calling the SINGLE CRUD TRANSACTION pattern, although used by many, hadn't been written up anywhere publicly. However, after submitting the pattern to *Overload* for review, it was pointed out that it is part of the FOREIGN KEY MAPPING pattern as written up by Martin Fowler [Fowler02]. I've just gone into a little more detail.

Name: Single CrUD Transaction

Intent

To create, update and delete items in a datastore within a single transaction.

Problem

Sometimes it's necessary to create, update and delete items in a datastore in a single transaction. Traditional web applications support create, update and delete in separate transactions and require the page to be reloaded between each action.

Modern web applications allow the items of a list to be created, updated and deleted in a browser without any interaction with the server or the underlying datastore. Therefore when the list is sent to the server side it must determine which items are new, which already exist and must be updated and which have been removed from the list and must be deleted.

Paul Grenyer is a husband, father, software consultant, author, testing and agile evangelist. He can be contacted at paul.grenyer@gmail.com

One simple solution is to delete all of the items from the datastore and simply replace them with the list of line items passed from the browser to the server. There are at least two potential drawbacks with this approach:

- If the datastore (such as a relational database) uses unique, numerical ids to identify each item in the list, the size of the ids can become very big, very quickly.
- If the datastore (such as a relational database) has other data which references the ids of the items in the list, the items cannot be deleted without breaking the referential integrity.

Solution

The `SINGLE CRUD TRANSACTION` pattern gets around these drawbacks by performing three operations within a single transaction:

- Delete all of the list items from the datastore whose ids are not in the list passed from the browser to the server.
- Update each of the items in the datastore whose ids match ids in the list passed from the browser to the server.
- Create new items in the datastore for each item in the list passed from the browser to the server which do not yet have ids.

Each action is executed within a single transaction so that if any individual action fails the list is returned to its original state.

Applicability

Use the SINGLE CRUD TRANSACTION pattern when:

- Datastores cannot have new items added, existing items updated and/or items removed in separate transactions.
- Creating new ids for each item in the list each time the datastore is modified is expensive or cumbersome.
- Removing all the items of a list from a datastore and recreating the list in the datastore breaks referential integrity.

Advantages and disadvantages

- Advantage: Entire update happens within a single transaction.
- Disadvantage: Three separate calls to the datastore within a single transaction. ■

References

[Alexander78] Christopher Alexander, 1978, *A Pattern Language: Towns, Buildings, Construction*, OUP USA (ISBN-13: 978-0195019193)

[Europlop] EuropPLoP (European Conference on Pattern Languages of Programs), http://www.europlop.net/

[Fowler02] Martin Fowler, 2002, *Patterns of Enterprise Application Architecture*, Addison Wesley, (ISBN-13: 978-0321127426)

[GoF94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design patterns: elements of reusable object-oriented software*, 1994, (ISBN-13: 978-0201633610)

Implementing Type-Classes as OCaml Modules

Type classes achieve overloading in functional paradigms. Shayne Fletcher implements some as OCaml modules.

Modular type classes

In this article, we revisit the idea of type-classes first explored in a previous blog post [Fletcher16a]. This time though, the implementation technique will be by via OCaml modules inspired by the paper 'Modular Type Classes' [Dreyer07] by Dreyer *et al*.

Starting with the basics, consider the class of types whose values can be compared for equality. Call this type-class **Eq**. We represent the class as a module signature.

```
module type EQ = sig
  type t
  val eq : t * t → bool
end
```

Specific instances of **Eq** are modules that implement this signature. Here are two examples.

```
module Eq_bool : EQ with type t = bool = struct
  type t = bool
  let eq (a, b) = a = b
end
module Eq_int : EQ with type t = int = struct
  type t = int
  let eq (a, b) = a = b
end
```

Given instances of class **Eq** (**X** and **Y** say,) we realize that products of those instances are also in **Eq**. This idea can be expressed as a functor with the following type.

```
module type EQ_PROD =
  functor (X : EQ) (Y : EQ) →
    EQ with type t = X.t * Y.t
```

The implementation of this functor is simply stated as the following.

```
module Eq_prod : EQ_PROD =
  functor (X : EQ) (Y : EQ) → struct
    type t = X.t * Y.t
    let eq ((x1, y1), (x2, y2))
      = X.eq (x1, x2) && Y.eq(y1, y2)
end
```

With this functor we can build concrete instances for products. Here's one example.

```
module Eq_bool_int :
  EQ with type t = (bool * int)
    = Eq_prod (Eq_bool) (Eq_int)
```

The class **Eq** can be used as a building block for the construction of new type classes. For example, we might define a new type-class **Ord** that admits types that are equality comparable and whose values can be ordered with a 'less-than' relation. We introduce a new module type to describe this class.

```
module type ORD = sig
  include EQ
  val lt : t * t → bool
end
```

Ad hoc polymorphism

In programming languages, there is a particular kind of polymorphism known formally called ad hoc polymorphism but better known as overloading. For example with overloading, an operator like + may be defined that works for many different kinds of numbers.

In the programming language Haskell, a language construction called *type classes* provides a structured way to provide for ad hoc polymorphism. The OCaml programming language does not have type classes but rather provides a construction called modules. Ad hoc polymorphism via Haskell-like typeclass style programming can be supported in OCaml by viewing type classes as a particular mode of use of modules. Indeed, the module approach can be argued as better in the sense that programmers can have explicit control over which type class instances are available in a given scope.

Here's an example instance of this class.

```
module Ord_int : ORD with type t = int = struct
  include Eq_int
  let lt (x, y) = Pervasives.( < ) x y
end
```

As before, given two instances of this class, we observe that products of these instances also reside in the class. Accordingly, we have this functor type

```
module type ORD_PROD =
  functor (X : ORD) (Y : ORD) → ORD with type t
    = X.t * Y.t
```

with the following implementation.

```
module Ord_prod : ORD_PROD =
  functor (X : ORD) (Y : ORD) → struct
    include Eq_prod (X) (Y)
    let lt ((x1, y1), (x2, y2)) =
      X.lt (x1, x2) || X.eq (x1, x2) &&
      Y.lt (y1, y2)
end
```

This is the corresponding instance for pairs of integers.

```
module Ord_int_int = Ord_prod (Ord_int) (Ord_int)
```

Here's a simple usage example.

```
let test_ord_int_int =
  let x = (1, 2) and y = (1, 4) in
  assert ( not (Ord_int_int.eq (x, y)) &&
    Ord_int_int.lt (x, y))
```

Shayne Fletcher is a programmer living in New York with 20 years experience, the last 5 of which in OCaml. He can be reached at shayne@shaynefletcher.org

The existence of the Show class is all that is required to enable the writing of our first parametrically polymorphic function

Using type-classes to implement parametric polymorphism

This section begins with the **Show** type-class.

```
module type SHOW = sig
  type t
  val show : t → string
end
```

In what follows, it is convenient to make an alias for module values of this type.

```
type 'a show_impl = (module SHOW with type t = 'a)
```

Here are two instances of this class...

```
module Show_int : SHOW with type t = int = struct
  type t = int
  let show = Pervasives.string_of_int
end
module Show_bool : SHOW with type t = bool
    = struct
  type t = bool
  let show = function | true → "True"
    | false → "False"
end
```

...and here these instances are 'packed' as values:

```
let show_int : int show_impl =
  (module Show_int : SHOW with type t = int)
let show_bool : bool show_impl =
  (module Show_bool : SHOW with type t = bool)
```

The existence of the **Show** class is all that is required to enable the writing of our first parametrically polymorphic function.

```
let print : 'a show_impl → 'a → unit =
  fun (type a) (show : a show_impl) (x : a) →
  let module Show =
    (val show : SHOW with type t = a) in
  print_endline@@ Show.show x
let test_print_1 : unit = print show_bool true
let test_print_2 : unit = print show_int 3
```

The function **print** can be used with values of any type **'a** as long as the caller can produce evidence of **'a**'s membership in **Show** (in the form of a compatible instance).

Listing 1 begins with the definition of a type-class **Num** (the class of additive numbers) together with some example instances.

The existence of **Num** admits writing a polymorphic function **sum** that will work for any **'a list** of values if only **'a** can be shown to be in **Num**.

```
let sum : 'a num_impl → 'a list → 'a =
  fun (type a) (num : a num_impl) (ls : a list) →
    let module Num =
      (val num : NUM with type t = a) in
    List.fold_right Num.( + ) ls (Num.from_int 0)
let test_sum = sum num_int [1; 2; 3; 4]
```

```
module type NUM = sig
  type t
  val from_int : int → t
  val ( + ) : t → t → t
end

type 'a num_impl = (module NUM with type t = 'a)

module Num_int : NUM with type t = int = struct
  type t = int
  let from_int x = x
  let ( + ) = Pervasives.( + )
end

let num_int = (module Num_int : NUM with
  type t = int)

module Num_bool : NUM with type t = bool = struct
  type t = bool

  let from_int = function | 0 → false
    | _ → true
  let ( + ) = function | true → fun _ → true
    | false → fun x → x
end

let num_bool =
  (module Num_bool : NUM with type t = bool)
```

Listing 1

This next function requires evidence of membership in two classes.

```
let print_incr : ('a show_impl * 'a num_impl)
    → 'a → unit =
  fun (type a) ((show : a show_impl),
    (num : a num_impl)) (x : a) →
    let module Num =
      (val num : NUM with type t = a) in
    let open Num
    in print show (x + from_int 1)
(*An instantiation*)
let print_incr_int (x : int) : unit
  = print_incr (show_int, num_int) x
```

If **'a** is in **Show** then we can easily extend **Show** to include the type **'a list**. As we saw earlier, this kind of thing can be done with an appropriate functor. (See Listing 2.)

There is also another way: one can write a function to dynamically compute an **'a list show_impl** from an **'a show_impl** (see Listing 3).

The type-class **Mul** is an aggregation of the type-classes **Eq** and **Num** together with a function to perform multiplication. (Listing 4.)

```
module type LIST_SHOW =
  functor (X : SHOW) →
    SHOW with type t = X.t list

  module List_show : LIST_SHOW =
    functor (X : SHOW) → struct
      type t = X.t list

      let show =
        fun xs →
          let rec go first = function
            | [] → "]"
            | h :: t →
              (if (first) then "" else ", ")
               ^ X.show h ^ go false t
          in "[" ^ go true xs
    end
```

Listing 2

A default instance of **Mul** can be provided given compatible instances of **Eq** and **Num**. (See Listing 5.)

Specific instances can be constructed as needs demand (Listing 6).

Note that in this definition of **dot**, coercion of the provided **Mul** instance to its base **Num** instance is performed.

Listing 7 provides an example of polymorphic recursion utilizing the dynamic production of evidence by way of the **show_list** function presented earlier. ■

This article was previously published as a blog post in 2016. [Fletcher16b] and the source is available at: https://github.com/shayne-fletcher/overload-2017/blob/master/mod.ml

```
let show_list : 'a show_impl → 'a list show_impl
   = fun (type a) (show : a show_impl) →
    let module Show =
      (val show : SHOW with type t = a) in
    (module struct
      type t = a list
      let show : t → string =
        fun xs →
          let rec go first = function
            | [] → "]"
            | h :: t →
              (if (first) then "" else ", ")
                 ^ Show.show h ^ go false t
          in "[" ^ go true xs
    end : SHOW with type t = a list)

let testls : string = let module Show =
  (val (show_list show_int)
  : SHOW with type t = int list) in
  Show.show (1 :: 2 :: 3 :: [])
```

Listing 3

```
module type MUL = sig
  include EQ
  include NUM with type t := t

  val mul : t → t → t
end

type 'a mul_impl = (module MUL with type t = 'a)

module type MUL_F =
   functor (E : EQ) (N : NUM with type t = E.t)
   → MUL with type t = E.t
```

Listing 4

```
module Mul_default : MUL_F =
  functor (E : EQ) (N : NUM with type t = E.t)
      → struct
    include (E : EQ with type t = E.t)
    include (N : NUM with type t := E.t)

    let mul : t → t → t =
      let rec loop x y = begin match () with
         | () when eq (x, (from_int 0))
           → from_int 0
         | () when eq (x, (from_int 1)) → y
         | () → y + loop (x + (from_int (-1))) y
       end in loop
end

module Mul_bool : MUL with type t = bool =
  Mul_default (Eq_bool) (Num_bool)
```

Listing 5

References

[Dreyer07] Derek Dreyer, Robert Harper and Manuel M. T. Chakravarty, 'Modular Type Classes', 2007, available online at http://www.cse.unsw.edu.au/~chak/papers/mtc-popl.pdf

[Fletcher16a] Shayne Fletcher, 'Haskell type-classes in OCaml and C++', available at http://blog.shaynefletcher.org/2016/10/haskell-type-classes-in-ocaml-and-c.html

[Fletcher16b] Shayne Fletcher, 'Implementing type-classes as OCaml modules', available at http://blog.shaynefletcher.org/2016/10/implementing-type-classes-as-ocaml.html

[Kiselyov14] Oleg Kiselyov, 'Implementing, and Understanding Type Classes', updated November 2014, available at http://okmij.org/ftp/Computation/typeclass.html


```
module Mul_int : MUL with type t = int = struct
  include (Eq_int : EQ with type t = int)
  include (Num_int : NUM with type t := int)
  let mul = Pervasives.( * )
end

let dot : 'a mul_impl → 'a list → 'a list → 'a
  = fun (type a) (mul : a mul_impl) →
    fun xs ys →
      let module M =
        (val mul : MUL with type t = a) in
      sum (module M : NUM with type t = a)
        @@ List.map2 M.mul xs ys
let test_dot =
  dot (module Mul_int : MUL with type t = int)
    [1; 2; 3] [4; 5; 6]
```

Listing 6

```
let rec replicate : int → 'a → 'a list =
  fun n x → if n <= 0
    then [] else x :: replicate (n - 1) x
let rec print_nested : 'a. 'a show_impl →
  int → 'a → unit = fun show_mod → function
  | 0 → fun x → print show_mod x
  | n → fun x → print_nested
    (show_list show_mod) (n - 1) (replicate n x)
let test_nested =
  let n = read_int () in
  print_nested (module Show_int : SHOW
  with type t = int) n 5
```

Listing 7

Evolutionary Computing Frameworks for Optimisation

Evolutionary algorithms can find optimal solutions to problems. Aurora Ramírez and Chris Simons give us an overview.

Within artificial intelligence, there is a category of programming problems where although the outcomes are known in terms of maximising or minimising some desired objective, the corresponding input values required to achieve this outcome are unknown. Such problems are known as optimisation problems, because solving the problem requires discovering an optimal solution in terms of either best quality (i.e. maximisation), or least cost (minimisation). Optimisation problems are found across a wide range of domains, e.g. from maximising energy outputs by optimising wind farm or generator placement, to minimising fuel costs by optimising delivery routes.

To address such optimisation problems in computing, a widely used approach involves firstly defining the space of all possible solutions to the problem. Then, with a programmatic encoding of all possible solutions representing a search space, it's possible to travel through the space, searching for optimal solutions.

Perhaps the simplest way to travel through the search space is to enumerate each possible solution. Such an approach is straightforward and uninformed insofar as it doesn't use any information about the problem domain to steer the search. For small scale search spaces, this approach can be highly effective. However, some search spaces can get very large, very quickly.

For example, in the 'travelling salesman problem', a number of cities are to be visited by a salesman. The salesman sets out to visit the cities in turn, returning to the starting point at the end of the journey, visiting each city once only. The goal of the salesman is to minimise travelling costs and CO_2 emissions by locating the path of least distance, i.e. the optimal path, around the cities. The number of possible solution paths, of course, depends on the number of cities to be visited. The greater the number the cities, the greater the number of possible solution paths. It just so happens that the number of possible solution paths is *n!* where *n* in the number of cities. For a few cities, e.g. 4, the number of possible solution paths is 24. However, for 100 cities, the number rises to approximately 10^{157}, while for 1000 cities, the number is roughly 10^{2567}. In fact, there are many other optimisation problems that show such scaling characteristics. Examples include many well-known allocation problems such as course timetabling, nurse rostering, process scheduling, network routing, vehicle delivery scheduling, and load balancing etc. For problems of such increasing scale, exhaustively examining each possible solution path is beyond reasonable computation time, and so alternative approaches are required. One alternative approach is inspired by the biology of the natural world around us, i.e. natural evolution.

Aurora Ramírez is a Computer Scientist and PhD student in the Department of Computer Science and Numerical Analysis at the University of Córdoba, Spain. Her research is focused on innovative search and optimisation techniques to support software engineers during software development, especially for architectural analysis of complex software systems. Contact: aramirez@uco.es

Chris Simons lectures at the University of the West of England, Bristol, in areas such as artificial intelligence and software development. Chris is interested in how software can learn from people, and vice versa, for mutual learning. Contact Chris at chris.simons@uwe.ac.uk

Evolutionary algorithms

In nature, there are two biological aspects at the heart of natural evolution. Firstly, selection of individuals for reproduction according to their fitness ensures that superior characteristics present in a population are more likely to survive in future generations. Secondly, sexual reproduction recombines the genetic information encoding parents' characteristics, resulting in offspring that are in some ways different to parents. In addition, gene mutation can also occasionally occur in nature. A mixture of recombination and mutation thus ensures a degree of variety in the population. Of course, all living things exist in environments that are prone to change. A combination of mechanisms for both selection and variety promotion enables a population of individuals, over time, to adapt to any changes in its environment.

Taking inspiration from natural biology, the notion of an evolutionary computing algorithm was suggested some time ago. Although the origin of the idea is not known for certain, Turing [Turing52] mentions the possibility of 'genetical programming' in his 1950 article when considering the question "can machines think?" Perhaps the first implementation of an evolutionary algorithm was developed by Fraser in 1957 [Fogel02], although this was an attempt to simulate the characteristics of species in natural evolution, rather than investigating computational optimisation. An early attempt at optimisation of the performance of finite state machines was described by Fogel et al. in 1966 [Fogel66] as 'evolutionary programming'. Subsequently, many proposals for evolutionary search and optimisation algorithms have emerged, and a recent article in *Overload* describes an evolutionary algorithm approach on how to 'program your way out of a paper bag' [Buontempo13].

The first task in applying an evolutionary algorithm for optimisation is to programmatically encode a representation of solution individuals. In this regard, evolutionary algorithms are quite flexible – virtually any representation may be used (although overly complex encodings can impact algorithm performance and impede understandability). Example representations can include, for example, vectors of 'genes' coding for solution characteristics, or where a better match to the problem domain can be found, perhaps encoding genes in graphs or tree structures. Because the solution representation codes for the 'genes' of the individual, the full expression of the genes is referred to by its biological meaning as a 'genotype'. It's important to draw on the characteristics of the problem domain to help formulate a solution representation.

The second task is to implement the evolution of a population of solution individuals over many generations. Listing 1 is a typical evolutionary outline approach, taken from [Eiben15].

As can be seen, there are a number of algorithm aspects to customise to the needs of the problem domain. For example, the termination condition `while(not done)` for the evolutionary loop could be the point at which

Recombination and mutation ensure variety in the offspring, which may be particularly necessary in complex optimisation problems where exploration of the search space is crucial.

```
initialise population at random
while( not done )
  evaluate each individual
  select parents
  recombine pairs of parents
  mutate new candidate individuals
  select candidates for next generation
end while
```

Listing 1

individuals of sufficient fitness have been arrived at, or when a certain computational budget has been exhausted. The evaluation of each solution individual is highly problem specific, and relates to the optimisation being conducted. For maximisation problems, measures of fitness are typically related to the required quality measures of the problem. For minimisation problems, measures of fitness typically relate to the cost or expense of solution individuals. Often, solution fitness depends on more than one quality/cost measure, in which case aggregation of individual fitness values to arrive at an overall evaluation of fitness may be necessary. The algorithm also includes a degree of randomness in that only a proportion of the population may be selected as parents for the next generation, and likewise for recombination and mutation.

Achieving population variety is achieved by recombining and mutating genetic information. Recombination and mutation ensure variety in the offspring, which may be particularly necessary in complex optimisation problems where exploration of the search space is crucial.

Evolutionary algorithms have been applied in a wide variety of optimisation problem domains. Further information on the application of evolutionary computing is widely available, although [Eiben15] is a compact and readable introduction. Given the relative maturity of evolutionary computing, it's perhaps not surprising that a number of frameworks have emerged to provide programmers with reusable components and interfaces for customisation to a variety of problem domain applications.

Evolutionary frameworks for optimisation

Evolutionary frameworks for optimisation allow rapid, exploratory prototyping with evolutionary algorithms by means of generic 'building blocks' (i.e. components and interfaces) [Parejo12] that, when properly configured or extended, address a domain-specific problem (e.g. such as the Travelling Salesman Problem). These building blocks usually correspond to the steps of an evolutionary algorithm. For example, there may be a component to create a population of solution individuals, another to select the best ones, others to produce new solution individuals via recombination and mutation, etc. [Gagné06]. In addition, frameworks often facilitate the execution and monitoring of the algorithm with general capabilities such as loading problem-domain specific information from configuration files, monitoring progress and generating reports.

In short, characteristics of evolutionary computing frameworks for optimisation include:

- adaptable search components to create customised implementations;
- mechanisms for the integration of problem-specific knowledge, such as problem constraints and fitness function(s);
- components to configure and monitor the execution, thus allowing the user to set any required execution parameter and visualise intermediate results;
- general utilities to conduct experiments, including batch processing and parallel execution; and
- designed with best practices such design patterns in mind.

The use of such frameworks can bring many advantages for the programmer seeking to implement optimisation programs. Firstly, coding effort greatly decreases and, to a certain extent, quality and correctness of code are both ensured. Additional utilities, such as benchmarks and graphic environments, might also be important to some users. In general, the communities behind the development of these frameworks provide us with complete, open-source programming environments that can also be easily integrated in external tools. Subject to the learning curve of the framework, it's possible to get up and running with evolutionary optimisation programs very quickly, especially when compared with programming an evolutionary algorithm from scratch.

However, there are also some challenges to using such a framework. One is that a considerable number are currently available, making it difficult to know which one best fits the needs of the problem domain. As is typical in optimisation problems, a unique global optimal solution may or may not exist, and so it's recommended that programmers trial a shortlist of frameworks and evaluate their performance for a specific problem. This leads to a second challenge, the learning curve. Some frameworks have been developed by various open source initiatives, including academics as part of their on-going research activities. Because of this, a few evolutionary frameworks may not be regularly maintained, and their documentation may not always be readily available and up-to-date. Moreover, some knowledge about the specific algorithm variants and the influence of setting their execution parameters might be also required to make the most of the evolutionary techniques. Finally, regarding the integration of these frameworks with existing tools, there may be some restrictions in terms of programming languages and platforms available.

Having said that, evolutionary algorithms have been applied to a wide range of problem domains, so implementations for many widely used programming languages are available. Table 1 lists some mature and popular frameworks written in C++, Java, C#, Python and Matlab, together with their latest available version, and the date of release for the latest version. Links to further details on each framework are available in the references section at the end of the paper.

C++ libraries such as Evolving Objects and OpenBeagle were early attempts to popularise the use of evolutionary algorithms, and precursors of more up-to-date developments. OptFrame and Evolutionary

One classic example of a selection mechanism is performing a tournament between solutions selected at random, wherein the best one wins the competition based on fitness

Language	Framework	Version	Date
C++	Evolutionary Computation Framework (ECF)	1.4.2	2017
	Evolving Objects (EO)	1.3.1	2012
	jMetalCpp	1.7	2016
	Mallba	2.0	2009
	Open Beagle	3.0.3	2007
	OptFrame	2.2	2017
	PaGMO	2.5	2017
	ParadisEO	2.0.1	2012
Java	Java-based Evolutionary Computation Research System (ECJ)	24.0, 25.0	2017
	Evolutionary Algorithms Workbench (EvA)	2.2.0	2015
	Java Class Library for Evolutionary Computation (JCLEC)	4.0	2014
	jMetal	5.3	2017
	Multi-Objective Evolutionary Algorithm (MOEA) Framework	2.12	2017
	Opt4J	3.1.4	2015
C#	GeneticSharp (A C# Genetic Algorithm Library)	On-going	2017
	HeuristicLab (A Paradigm-Independent and Extensible Environment for Heuristic Optimization)	3.3.14	2016
Python	Distributed Evolutionary Algorithms in Python (DEAP)	1.1.0	2017
	jMetalPy	On-going	2017
	Pyevolve	0.6rc_1	2015
	PyGMO	On-going	2017
	Pyvolution	1.1	2012
Matlab	Genetic and Evolutionary Algorithm Toolbox for Matlab (GEATbx)	3.8	2017
	Global Optimisation Toolbox	R2017b	2017
	Matlab Platform for Evolutionary Multi-objective Optimisation (PlatEMO)	1.3	2017

Table 1

Computation Framework (ECF) are good active examples of this, both supporting parallelism via MPI (Message Passing Interface). OptFrame also provides implementations of the MapReduce paradigm, whereas the particular strength of ECF is straightforward configuration with few parameters. jMetal is a popular framework, and is the only framework

```
public abstract class PopulationAlgorithm
   extends AbstractAlgorithm{
  doInit(): void
  doSelection(): void
  doGeneration(): void
  doReplacement(): void
  doControl(): void
}
```

Listing 2

available in three different programming languages. Originally coded in Java, it is focused on implementations of evolutionary algorithms to solve multi-objective problems. MOEA Framework is another recent library to solve this type of problem, with well-documented and tested code. The most mature Java library is the Java-based Evolutionary Computation Research System (ECJ), which offers a great variety of search algorithms. Although perhaps less known, Java Class Library for Evolutionary Computation (JCLEC) provides extensible modules for more advanced techniques including machine learning, whereas Evolutionary Algorithms Workbench (EvA) and Opt4J include basic graphical user interfaces. However, the framework with the most complete graphical environment is probably HeuristicLab, developed under the .NET framework but also compatible with Linux systems. Finally, new developments are appearing for Matlab and Python, and among those worthy of mention is the PyGMO/PaGMO project, initially developed by the European Space Agency. PaGMO has been recently rebuilt to comply with language features of C++14 and 17.

Application example

The high-level programming interface of the Java Class Library for Evolutionary Computation (JCLEC) [Ventura08] is a representative example of an evolutionary optimisation framework. This open source library also offers extension modules to develop multi-objective algorithms and machine learning approaches, as well as a visual environment to run experiments.

Every evolutionary algorithm in JCLEC extends the class `PopulationAlgorithm`, which defines the main steps of the evolutionary process as shown in Listing 2.

To initialise the population, a component named `Provider` is invoked with the number of solutions to be created as a parameter. Classes extending the interfaces `IIndividual` and `ISpecies` are those defining the specific characteristics of the optimisation problem. On the one hand, each solution individual should contain a fitness object representing its quality. For the travelling salesman problem, an individual represents a possible route, while the fitness quality measure is the total distance to be minimised. General methods to copy and compare solutions are also required along the search process. The species provides additional information of the problem and how it is encoded. In our example, this element will contain the number of cities. Therefore, the

The next step is to generate new solutions by recombining and mutating the chosen ones. Again, independent components will be coded to perform each task.

```
public interface IIndividual{
  getFitness(): IFitness
  setFitness(IFitness): void
  copy(): IIndividual
  equals(Object): boolean
}

public interface IProvider{
  provide(int): List<IIndividual>
}

public interface ISpecies{
  createIndividual(T[]): IIndividual
}
```

Listing 3

species can create random routes, which are stored as permutations of the cities. (See Listing 3.)

A further component evaluates the quality of random solutions after their creation. The interface **`IEvaluator`** defines methods to evaluate a list of individuals, counts the number of evaluations (which might be a stopping criterion) and adapts the comparator to the specific problem. This way, individuals can be compared in terms of their fitness values for both maximisation and minimisation problems. The evaluation of the travelling salesman problem will assess the distance between the consecutive cities on the route, as defined by the permutation representing the solution. The total distance is to be minimised.

For clearly defined optimisation problems such as the travelling salesman problem, the quality of the solution is quantified using a **`double`** floating-point value. In this case, the interface **`IValueFitness`** provides the necessary methods to keep and retrieve the computed value, as well as additional methods to check if the value 'is good enough' against some criteria, and make copies. These two latter methods are inherited from the more general interface **`IFitness`**. (See Listing 4.)

```
public interface IEvaluator{
  evaluate(List<IIndividual>): void
  getNumberOfEvaluations(): int
  getComparator(): Comparator<IFitness>
}

public interface IValueFitness extends IFitness{
  getValue(): double
  setValue(double): void
  isAcceptable(): Boolean
  copy(): IFitness
}
```

Listing 4

```
public interface ISelector{
  select(List<IIndividual>)
    : List<IIndividual>
  select(List<IIndividual>, int)
    : List<IIndividual>
  select(List<IIndividual>, int, boolean)
    : List<IIndividual>
}

public interface IRecombinator{
  recombine(List<IIndividual>): List<IIndividual>
}

public interface IMutator{
  mutate(List<IIndividual>): List<IIndividual>
}
```

Listing 5

Once the population has been created, each generation of the evolution is performed by sequentially invoking the following methods (see Listing 2): **`doSelection( )`**, **`doGeneration( )`**, **`doReplacement( )`** and **`doControl( )`**. For selection, the most common approach is to delegate the functionality to a 'selector', a class implementing the interface **`ISelector`**. In JCLEC, diverse selection methods can be implemented via polymorphism as shown in the following listing. More specifically, the first method just picks as many solutions as there is in the received list (repetition is allowed by default), the second specifies the number of solutions to be returned, and the third one also allows the programmer to indicate whether repetition is permitted. One classic example of a selection mechanism is performing a tournament between solutions selected at random, wherein the best one wins the competition based on fitness.

The next step is to generate new solutions by recombining and mutating the chosen ones. Again, independent components will be coded to perform each task, using the interfaces **`IRecombinator`** and **`IMutator`** as reference. In their most simple form, both genetic operators receive a list of individuals and return another one with the modified solutions. Specific classes in JCLEC extend this idea to support the association of probabilities and transform the solutions depending on how the solutions are encoded. For the travelling salesman problem, a recombinator can interchange subroutes of two solutions to produce new ones. Mutation might just swap two cities for a given route. (See Listing 5.)

Relationships between the various interfaces offered by JCLEC are as shown in Figure 1.

Both user-defined classes or existing ones can be combined in JCLEC to run an evolutionary algorithm. To put all the components together, a configuration file in XML format specifies which class implements each component. JCLEC uses Java Reflection to instantiate these classes 'on the fly' during the configuration process, and then launches the execution.

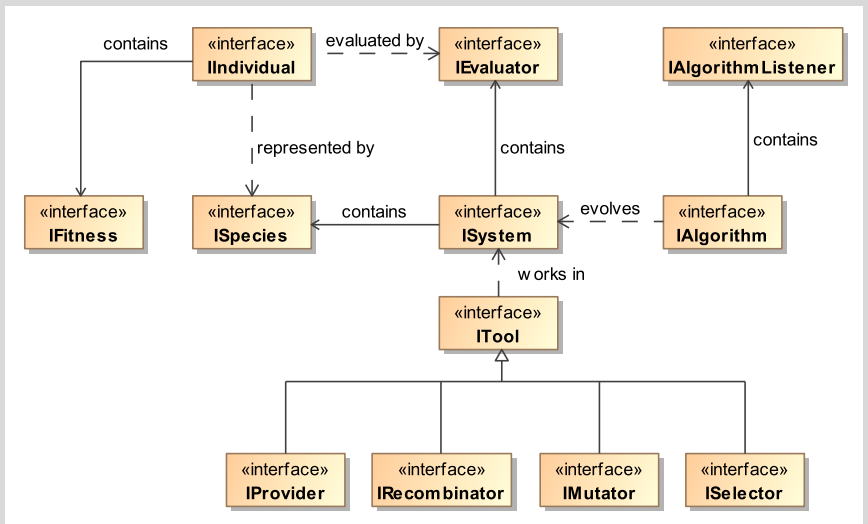


Figure 1

Listeners may be added to generate intermediate reports at a desired frequency, for example, at a specified number of evolutionary generations.

Taking the travelling salesman problem as an example, one possible resulting configuration is shown below. In this experiment, the evolving population comprises 100 solution individuals, and the termination criterion is 10000 fitness evaluations. The problem to be optimised in this experiment relates to 52 locations in the city of Berlin, and the specification of the 52 locations is held in a file `berlin52.tsp`. The genes of the solution genotype are encoded as an ordered array, containing 52 ordered elements – each element identifying a single location. The evaluation of each solution individual involves computing the sum of the distance between each location in the order specified in the genotype. In evolutionary computing terms, the total distance is the 'fitness' of the individual. In this experiment, optimisation involves minimising the distance to arrive at the least cost solution.

Parent individuals are selected by 'tournament' selection, i.e. two individuals are placed in a tournament where the single fittest wins. The probability that two selected parents will be recombined is 0.9, while the probability that recombined individuals will be further mutated is 0.2. Finally, intermediate reports are generated at every 10 generations of evolution (see Listing 6).

After running the evolutionary optimisation experiment for the travelling salesman problem with the above configuration, the JCLEC framework provides information of the optimisation, as shown in Figure 2.

After 1000 generations of evolution, the best and worst solution individuals in the population are shown. The best solution individual has a fitness value of 9483.023, i.e. the distance of the path in metres. The genotype reveals the travelling order of the locations (expressed as integer identifiers) resulting in this distance. The worst solution individual has a fitness value of 12737.531. Because the population of individuals retains some diversity (even after 1000 generations of evolution), additional information about the population is also provided.

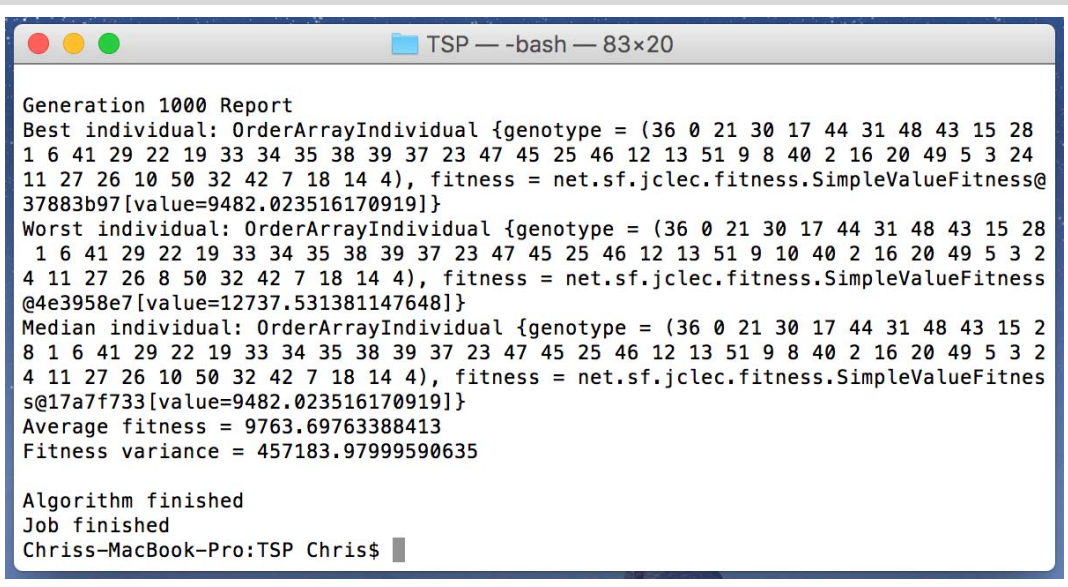


Figure 2

Information related to the median individual in the population is also made available, as is the average fitness of the population, and population fitness variance.

Practicalities and final thoughts

There are a number of practicalities relating to the application of evolutionary optimisation, including:

- Domain specific information relating to the optimisation problem is essential. For example, some notion of maximisation or minimisation is necessary to implement fitness-based selection. Also, an understanding of appropriate solution characteristics is necessary to encode a satisfactory presentation of the genetic information.
- Evolutionary optimisation is only appropriate for larger scale problems. If the scale of the solution search space is such that exhaustive enumeration of all individuals is possible, then this is preferable because we can be sure that the optimal solution has been discovered.
- Since evolutionary optimisation incorporates a degree of randomness (in selection and diversity preservation), it's not possible to prove that the 'best' solution(s) discovered are, in fact, optimal. However, sometimes just getting something that's 'good enough' can be great, improving the quality of the problem solution in ways otherwise not possible.
- Benchmarking evolutionary optimisation performance can be tricky. Firstly, although some comparative studies of optimisation frameworks exist (e.g. [Parejo12]), standard benchmark problem instances are less readily available. Secondly, customisation of optimisation components and parameters must be consistent across different problem domains and frameworks to ensure a fair comparison. Thirdly, because of the degree of randomness is present in the evolutionary frameworks, results can differ over different evolutionary 'runs' for the same optimisation problem. In this case, it's useful to execute many evolutionary optimisation 'runs' for the same problem, and perform statistical analysis if appropriate.
- What happens after a population has evolved? It's possible for complex optimisation problems that at algorithm termination, a population of equally optimal solutions has been discovered, but many individuals are dissimilar. At this point, programmer preference and judgement might be required to choose among the available solutions.

```
<experiment>
  <process algorithm-type="net.sf.jclec.algorithm.classic.SGE">
    <rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="123"/>
    <population-size>100</population-size>
    <max-of-evaluations>10000</max-of-evaluations>
    <species type="net.sf.jclec.orderarray.OrderArrayIndividualSpecies" genotype-length="52"/>
    <evaluator type="tutorial.TSP" file-name="berlin52.tsp" number-cities="52"/>
    <provider type="net.sf.jclec.orderarray.OrderArrayCreator"/>
    <parents-selector type="net.sf.jclec.selector.TournamentSelector">
      <tournament-size>2</tournament-size>
    </parents-selector>
    <recombinator type="net.sf.jclec.orderarray.rec.OrderPMXCrossover" rec-prob="0.9" />
    <mutator type="net.sf.jclec.orderarray.mut.Order2OptMutator" mut-prob="0.2" />
    <listener type="net.sf.jclec.listener.PopulationReporter">
      <report-frequency>10</report-frequency>
      <report-on-file>true</report-on-file>
    </listener>
  </process>
</experiment>
```

Listing 6

However, even with these practicalities in mind, evolutionary optimisation frameworks offer significant 'off-the-shelf' optimisation capabilities for programmers. Given their on-going development and increasing maturity, they can be attractive option for programmers working with large scale optimisation problems. ■

References and resources

[Buontempo13] 'How to Program Your Way out of a Paper Bag using Genetic Algorithms', F. Buontempo, *Overload*, December 2013, https://accu.org/index.php/journals/1825

[DEAP] DEAP: Distributed Evolutionary Algorithms in Python. https://github.com/DEAP (Last accessed: 23/10/17).

[ECF] ECF – Evolutionary Computation Framework. http://ecf.zemris.fer.hr/ (Last accessed: 23/10/17).

[ECJ] ECJ – A Java-based Evolutionary Computation Research System. https://cs.gmu.edu/~eclab/projects/ecj/ (Last accessed: 23/10/17).

[Eiben15] *Introduction to Evolutionary Computing*, 2nd Ed., A.E. Eiben, and J.E. Smith, Springer, 2015.

[EO] Evolving Objects (EO): an Evolutionary Computation Framework. http://eodev.sourceforge.net/ (Last accessed: 23/10/17).

[EvA] EvA2 – A Java based framework for Evolutionary Algorithms. http://www.ra.cs.uni-tuebingen.de/software/eva2/ (Last accessed: 23/10/17).

[Fogel02] 'In Memoriam: Alex S. Fraser', D. Fogel, *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 429–430, 2002.

[Fogel66] *Artificial Intelligence through Simulated Evolution*, L.J. Fogel, A.J. Owens, and M.J. Walsh, Wiley, 1966.

[Gagné06] 'Genericity in evolutionary computation software tools: principles and case-study', C. Gagné, M. Parizeau, *International Journal of Artificial Intelligent Tools*, vol. 15, no. 2, pp. 173–194. 2006.

[GEATbx] GEATbx – The Genetic and Evolutionary Algorithm Toolbox for Matlab. http://www.geatbx.com/ (Last accessed: 23/10/17).

[GenSharp] GeneticSharp. https://github.com/giacomelli/GeneticSharp (Last accessed: 23/10/17).

[GOT] Global Optimisation Toolbox. https://www.mathworks.com/products/global-optimization.html (Last accessed: 23/10/17).

[HL] HeuristicLab – A Paradigm-Independent and Extensible Environment for Heuristic Optimization. https://dev.heuristiclab.com/trac.fcgi/ (Last accessed: 23/10/17).

[JCLEC] JCLEC – A Java Class Library for Evolutionary Computation. http://jclec.sourceforge.net/ (Last accessed: 23/10/17).

[jMetal] jMetal: a framework for multi-objective optimization with metaheuristics. https://github.com/jMetal (Last accessed: 23/10/17).

[Mallba] MALLBA Library. http://neo.lcc.uma.es/mallba/easy-mallba/index.html (Last accessed: 23/10/17).

[MOEAFram] MOEA Framework – A Free and Open Source Java Framework for Multiobjective Optimization. http://moeaframework.org/ (Last accessed: 23/10/17).

[OB] OpenBeagle – A generic C++ framework for evolutionary computation. https://github.com/chgagne/beagle (Last accessed: 23/10/17).

[Opt4J] Opt4J – A Modular Framework for Meta-heuristic Optimization. http://opt4j.sourceforge.net/ (Last accessed: 23/10/17).

[OptFrame] OptFrame. https://sourceforge.net/projects/optframe/ (Last accessed: 23/10/17).

[Pagmo] Pagmo – The C++ Scientific Library for Massively Parallel Optimization. https://esa.github.io/pagmo2/ (Last accessed: 23/10/17)

[Paradiseo] Paradiseo – A Software Framework for Metaheuristics. http://paradiseo.gforge.inria.fr/ (Last accessed: 23/10/17).

[Parejo12] 'Metaheuristic Optimization Frameworks: A Survey and Benchmarking', J.A. Parejo, A. Ruiz-Cortés, S. Lozano, and P. Fernandez, *Soft Computing*, vol. 16, no. 3, pp. 527–561, 2012.

[PlatEMO] PlatEMO – Evolutionary multi-objective optimization platform. http://bimk.ahu.edu.cn/index.php?s=/Index/Software/index.html (Last accessed: 23/10/17)

[Pyevolve] Pyevolve. http://pyevolve.sourceforge.net/ (Last accessed: 23/10/17)

[Pygmo] PyGMO – The Python Scientific Library for Massively Parallel Optimization. https://esa.github.io/pagmo2/ (Last accessed: 23/10/17)

[Pyvolution] Pyvolution – A Pure Python Evolutionary Algorithms Framework. https://pypi.python.org/pypi/Pyvolution (Last accessed: 23/10/17)

[Turing52] 'Computing Machinery and Intelligence', A.M. Turing, *Mind*, vol. 59, no. 236, pp. 433–460, 1950.

[Ventura08] 'JCLEC: a Java framework for evolutionary computation', S. Ventura, C. Romero, A. Zafra, J.A. Delgado, C. Hervás, *Soft Computing*, vol. 12, no.4, pp. 381–392, 2008.

Afterwood

Tabs are controversial. Chris Oldwood reminds us of their many guises.

There can't be many three-letter words that can cause the average programmer to break out into a cold sweat, but the word 'tab' must surely feature fairly highly among them. I can imagine the word 'for' would cause a functional programming evangelist to give a little shudder given their dislike for explicit iteration, and the abbreviation 'int' must surely generate a lengthy sigh from anyone who's had to write cross-platform code in C or its closest brethren, C++. However, 'tab' cuts across paradigms and languages and holds a special place in our psyche.

A rather popular sci-fi TV programme tells us that 'space' is the final frontier and, for typists around 1900, that was certainly a major challenge they faced. If you think positioning text with CSS can be arduous at times try laying out a table of data on an old-fashioned typewriter. Repeatedly pressing the space bar and backspace keys to line up the cursor is going test anyone's patience, and that's before you factor in the effort each keystroke requires on such a mechanical device. RSI is an initialism so it's exempt from this particular discourse but the tab key must be congratulated here on its stress reducing measures.

It's funny to think that something we can't actually see would cause us programmers so much distress and yet, like a black hole, it has the ability to distort that which surrounds it. How many times have you inserted a space early in a line only for the remainder to spring exponentially further to the right as a gaping chasm opens up from nowhere; it's like a kerning disaster on a grander scale. They also say beauty is in the eye of the beholder but no one in their right mind would choose to use 8 spaces for indentation, but apparently, historically, that's what the world once thought was considered Good Taste™.

My first foray into the world of Python left me with a scar I still bear to this very day and which came courtesy of the invisible enemy. After starting my first non-trivial Python program in the (then) new-fangled IDLE Python IDE, I quickly switched to the more comfortable Visual Studio text editor to finish my sysadmin masterpiece. Sadly I wasted the following hour and a half scratching my head and debugging until I discovered the cruel lesson from the school of hard knocks about mixing tabs and spaces in the same Python block. I had forgotten that back then in Washington the space was out of favour, there, the acronym TARDIS probably stood for Tabs Are Redmond's Default Indentation Style, and so lines of Python code were being skipped or executed in a seemingly arbitrary fashion. Goto Fail, do not pass Go.

Of course tabs are bad for you. Anyone who grew up in north-east England knew that because tabs are a slang word for cigarettes. If you believe the oft-quoted 10 lines of code per developer per day, then, assuming a low degree of nesting you should easily keep your habit under 20-a-day, but the rock-star programmers are probably chain-smoking their way to a very early grave. And that's before we've even considered their late-night coding antics where they might drop a tab or two of some illicit drug to fuel their spree. To them an ACID guarantee is an entirely different prospect.

Unsurprisingly I approached F#, another whitespace-sensitive language, with a little more trepidation. However, learning from the mistakes of their forefathers, the F# designers just banned tabs by default, which seems very sensible. If you really want to shoot yourself in the foot and use tabs you can, but you need a special disclaimer at the top of every file to waive your rights to any form of support on Stack Overflow. It's not just your life you're wrecking – friends don't let friends use tabs, ever.

Just when you think the eternal war between tabs and spaces is beginning to decline as the ever growing list of vets speaks out over the folly of choice and the remaining battles diminish to a few skirmishes on the outer rim of the Internet, a new programming language comes along and tips petrol over the smouldering embers. Google's Go is a highly opinionated language that's not afraid to upset anyone's apple cart, and the de facto choice of using tabs for indentation is just one major example of this. Actually, that's not entirely true: the language itself is just as tolerant of ugly code as any other – it's the community that's decided to embrace a common standard. On matters of style the bundled go fmt tool is judge, jury and re-formatter. Expect your pull requests to become bantha fodder if your whitespace is not whiter than white.

This isn't the first time a large corporation has tried to sell the tab to a sceptical market. Back in the late 1970s, the Coca-Cola Company brought TaB to the UK in the hope of capturing a slice of the diet soda market. It was rebranded TaB Clear before I was old enough to appraise it, but it didn't last long – disappearing along with Cadbury's Fuse. I guess the buying public wants to see something for their money.

It's not all doom-and-gloom for the humble tab. In the early 2000s, as the Internet was really taking off big-time, the modern developer found themselves in a new kind of hell. The ever-growing Internet started to become the predominant source of developer information and so it was not uncommon to have multiple browser instances open, switching back-and-forth as we pieced together the answers to our puzzle. Although Opera sported an MDI style UI it was not really until the new mainstream contender to the browser throne – Firefox – appeared that the browser tab really took off and reduced the number of task bar icons back to a sustainable cognitive load.

Although heralded as the saviour of the browser, the tab as a UI element was not without its detractors. The tab fad really started to take off in the early 90s as the Office suites began to look for alternatives to MDI and cluttered configuration dialogs. The Interface Hall of Shame has plenty of examples of where the wheels fell off. For the browser, however, the real elephant in the room was a poor security record and a need to isolate each web site into its own process. Suddenly each new tab meant another multi-megabyte process and ever more strain on the humble computer. But it isn't just browsers; even native apps like Slack with essentially a tab per team are using process-level isolation and hogging resources; it's not uncommon to find Chrome and Slack duking it out behind the scenes on a developer's laptop apparently striving for global domination.

So, all tabs are evil, right? Well maybe not, there is one category of tab that actually seems to be useful and pretty much side-effect free – guitar tabs. For those of us whose only musical ability stems from playing Guitar Hero, the guitar tab provides a lifeline for learning to play the classics without having to grok some ancient music notation first. Maybe the tab has a stairway to heaven after all. ■

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology in plush corporate offices. He also commentates on the Godmanchester duck race and can be easily distracted via gort@cix.co.uk or @chrisoldwood

JOIN THE ACCU!

You've read the magazine, now join the association dedicated to improving your coding skills.

The ACCU is a worldwide non-profit organisation run by programmers for programmers.

With full ACCU membership you get:

- 6 copies of *C Vu* a year
- 6 copies of *Overload* a year
- The ACCU handbook
- Reduced rates at our acclaimed annual developers' conference
- Access to back issues of ACCU periodicals via our web site
- Access to the *mentored developers projects*: a chance for developers at all levels to improve their skills
- Mailing lists ranging from general developer discussion, through programming language use, to job posting information
- The chance to participate: write articles, comment on what you read, ask questions, and learn from your peers.

Basic membership entitles you to the above benefits, but without Overload.

Corporate members receive five copies of each journal, and reduced conference rates for all employees.

How to join

You can join the ACCU using our online registration form. Go to **www.accu.org** and follow the instructions there.

Also available

You can now also purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG