overload 145

JUNE 2018 £3

A Short Overview of Object Oriented Software Design

Object oriented design has many principles. We demonstrate good design through a role playing game.

Using {fmt} For Tracing

Using {fmt} to trace compound and custom types

How to Write a Programming Language: Part 1, The Lexer

We start writing a simple programming language from scratch

Afterwood

The curse of Agile

A magazine of ACCU

ISSN: 1354-3172

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

OVERLOAD 145

June 2018

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 146 should be submitted by 1st July 2018 and those for Overload 147 by 1st September 2018.

Overload is a publication of the ACCU

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

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.

Automate all the things

Automation can speed things up. Frances Buontempo considers how it can make things worse.

Preparing for the ACCU conference and making a few tweaks to my up-coming book in-line with some feedback, I haven't had a chance to write an editorial. I still dream of creating an automatic editorial generator, but need to learn a lot more about natural language processing to make any progress with that. So what has been eating my time?

At work, I am trying to rejuvenate and repurpose some old FORTRAN code. Originally, it was used to model software reliability. In fact, there are several models with the subtle differences in approach, but on a high level, they each take a list of event times and use these to predict when future events might happen. In terms of software reliability, these events mean finding a bug in a code base, or other fault, and fixing it. Realists may say you can't find and fix a bug instantaneously. In modelling terms, you should never let reality get in the way, so could use the time when the bug fix is committed to the code base instead. If the event times are getting further apart, things are improving. In theory, you can fit these to some statistical models and decide how likely it is that all the bugs have been removed from a code base. Overviews are available, for example, *Reliability Growth: Enhancing Defense System Reliability* has a chapter giving an overview of the models [NRC]. Our team is trying to find ways to apply these models to cybersecurity. The theory is a security problem is an event with a timestamp, so sending in event times to the models allows you to predict when the next events might happen.

I have eight models, with over fifteen data sets along with previously generated output files for some of these inputs. There are no unit tests, which is no surprise, however the expected outputs for given inputs provide a way to test the code. I wrote a script to build the models and run them against all the inputs. When I ran this, several of the models exploded in various exciting ways. My instinct to automate this first left me drowning in error messages. A smaller script to check how many input files have corresponding outputs would have been revealing. Several inputs did not having matching outputs. It turns out those without outputs were crashing and causing my well-intentioned script to fail. Once I limited the inputs to those that didn't explode, I ended up with different formats in the output, so a simple diff wouldn't work. Queue another script to compare files of numbers. The generated files have rows of numbers for each event. The original files have blocks of up to four numbers on a row, so you need a bit of guess work to figure out when you are actually on a new event. The numbers don't match exactly either, which will need a meeting to discuss. I haven't yet checked running the same code on the same input provides the same output. That's another story. The highlight here is writing scripts to automate this seemed like a good idea. If I had tried one or two things by hand first, I may have noticed the missing outputs for some of the inputs and actually made progress quicker by slowing down. I may have found my superhero name by doing this. I recently found out that 'kilogirl' was an early unit of computing power, equivalent to a thousand hours of manual computing labour. This seems to come from a book called *Broad Band: The Untold Story of the Women Who Made the Internet* by Claire L Evans [Evans18]. Manually comparing the columns in the output files would probably be a thousand hours. Kilogirl it is.

At the recent ACCU conference, Daniele Procida advocated a Zen approach to problem solving, saying "Don't just do something. Sit there." [Procida18]. Charging straight in without thinking first is often asking for trouble. In *Overload* 128, I considered what you should automate. [Buontempo15]. The origins of computers and fear of machines taking over the world, or at least taking people's jobs is old. And yet, history repeats itself. And indeed, my errors of signalling NaNs also repeated themselves when I ran my automatic script. If I had listened to my own advice, I would have tried one input at a time for a while to get a feel for the unfamiliar code base and file format, before writing a script. Automate all the things, but not yet.

I am not the only person wondering if I have too much automation. A recent article claims Elon Musk regards too much automation at Tesla Inc as a mistake [Hull18]. He says, "We had this crazy, complex network of conveyor belts, and it wasn't working, so we got rid of that whole thing." My unfamiliar FORTRAN code base feels like that and I am tempted to ditch it and re-write it in another language. I might do that as a learning exercise; however, my mission is to make this work, so can't get rid of the whole thing. Musk also said, "Humans are underrated" [Musk18] in response to the discussion. Automation is supposed to save humans form error-prone, boring repetitive tasks that machines are more suited for, however the article points out, automation can be:

> Expensive and is statistically inversely correlated to quality. One tenet of lean production is 'stabilize the process, and only then automate.' If you automate first, you get automated errors.

You always need to think about what you are trying to achieve and measure to check this is happening. A brief discussion around this [Malone18] was nailed on the head:

> I think they're saying too much automation too soon is an expensive mistake that the Germans and Japanese have learned. But they still automate when products mature.

Some of us know more about the manufacturing processes than others, but most of us have encountered interviews. Can that process be automated? Have you ever have been sent a Hackerrank or Codility test as a pre-screen? Or a coding exercise that you suspect gets sent through a test suite? These take time; you are often given 90 minutes to complete the online exercises. Do you have 90 minutes spare where you can be certain no one will interrupt? Maybe, maybe not. What happens if you fail the automatic tests? Furthermore, why don't these websites allow you to use a test suite? Or version control? Whatever the reason, this means writing code in a strange way for many of us. Some companies will still look at your submissions even if you don't get 100%, and use your solutions as a basis for discussion. There's a tension between companies

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.

avoiding wasting their employees' time when recruiting and them wasting interviewees time. There's also a need to make sure the process is 'fair' by ensuring everyone has been asked the same questions. On paper that sounds sensible, but different people have different skill sets. Some people are quicker than others too. I was quicker when I was younger, and am beginning to wonder how much of the lack of diversity in some organisations is down to their recruitment process. Do you want a quickly coded, hand-rolled algorithm, or easy-to-follow code in version control complete with unit tests? Or someone who is good at bug hunting? Or team building and mentoring? How would you test these skills automatically?

Before you get as far as being tested, you usually submit a CV. A script can search for skills in this, which leads to suggestions of hiding buzz words in white on a white background to get picked out by the bots. Automatic CV screening can exclude people who may not have followed a conventional route. I do not have a Comp. Sci. degree so some automatic process will instantly exclude me. Some may insist on straight As at school. I got half As and half Cs. Again, #fail. I do have a PhD, but an automated process may not value this. Some people do not have a degree but can still code.

I understand that some British universities are dropping a requirement for 'A' level physics as a prerequisite to study engineering, in order to encourage diversity. The theory is that some girls avoid physics classes since they would be the only female in the room. That doesn't mean they aren't interested or capable, just that they don't want to be the odd one out. One gentleman, prior to a diversity and inclusion in STEM talk I attended, informed me that women aren't interested in science. He was partially right; many men aren't in the slightest bit interested either. If women instantly get excluded in the UK because they can't face being the odd one out at school, claiming they are not interested enough in the subject is missing the point. Our bias and assumptions *do* filter into our processes and can make the situation worse if the process is automated. As Musk said, humans are underrated. Sometimes. Ms Teedy Deigh mentioned 'BIBO'; bias in, bias out, in her article for our last issue [Deigh18]. I used this phrase in my keynote to the ACCU conference last year, so am pleased she was there listening. I am always amused that feedforward neural networks usually have a bias neuron. Its purpose is to allow all zero inputs to map to non-zero outputs, or more generally shift the inputs up or down [Stackoverflow]. Yet there it is, right in the middle of a common AI technique: bias, built in.

A recent post on Medium by Michael Jordan considers AI [Jordan18]. He tells a concerning tale of his pregnant wife's ultrasound test revealing markers for Down syndrome. They were told the risk of the baby having the syndrome had increased to 1 in 20, but the risk of an amniocentesis test killing their baby was 1 in 300. Being a statistician, he dug into the numbers, and concluded the increased number of pixels on current machines meant the health care officials might be modelling white noise, since the original numbers were based on white spots showing up on older displays. Using old data and models as the world changes, especially if you bring automation into the mix, is asking for mistakes. He notes that before civil engineering, people still built bridges and some collapsed. He sees the need for a new engineering discipline around AI, requiring human perspectives. He notes some early neural networks were used to optimise the thrust of the Apollo spaceships and are now used to power decisions Amazon, Facebook and other large tech companies make. Amazon didn't realise the London Marathon runs along the end of my road, so their clever logistic algorithm didn't pick a different day to make a delivery for me. Sometimes, humans need to call out that new information has changed the situation. Furthermore, a narrow set of voices might be causing bias in the topics researched and the outcomes. He says, "There is a need for a diverse set of voices from all walks of life, not merely a dialog among the technologically attuned. Focusing narrowly on human-imitative AI prevents an appropriately wide range of voices from being heard." He ends by saying, "Let's broaden our scope, tone down the hype and recognize the serious challenges ahead."

AI seems like the ultimate way to automate all the things, but various the scripts and tooling we write ourselves takes us part-way there. My scripts to attempt to verify the modelling code are one small, slightly failed part. Online coding tests to filter candidates are arguably another. Automating a build is more sensible. Having scripts to deploy releases is wise. Encouraging a QA department to automate some of the process is a good idea. Automating all the things can slow you down and introduce bias. Inclusivity and diversity matter. This means different things to different people. Different countries have differing problems. One keynote at this year's conference was about diversity and inclusivity. It became apparent, at least to me, that attendees from some countries don't believe there is a problem with few women studying or involved with STEM. A lightning talk by Robert Smallshire showed the variation between countries, in particular mentioning 49% of STEM students being women and shared a graph of the global index gender gap against the percentage women among STEM graduates [Sossamon18]. What happens after graduation is another matter [UNESCO]. We need to be careful about context and assumptions when we talk to each other. There is a problem in the UK. Particularly with guys, and I mean guys, turning up to inclusivity talks telling me women don't like tech subjects.

Automating everything is asking for trouble. Being aware of context and doing some fact finding is important. The ACCU conference also mentioned the Include CPP group [IncludeCpp]. I've joined their discord chat channel, which is another excuse for failing to write an editorial. However, if you want people to chat to and some support or encouragement, do get involved.

References

[Buontempo15] Frances Buontempo (2015) 'Semi-automatic Weapons', Overload 128, Aug 2015 https://accu.org/index.php/journals/2133

[Deigh18] Teedy Deigh (2018) 'Ex Hackina', *Overload* 144, https://accu.org/index.php/journals/2484

[Evans18] Claire L. Evans (2018) *Broad Band: The Untold Story of the Women Who Made the Internet*, Portfolio, ISBN 9780735211759

[Hull18] Dana Hull (2018) 'Musk Says "Excessive Automation Was My Mistake"', Bloomberg 13 April 2018, https://www.bloomberg.com/news/articles/2018-04-13/musk-tips-his-tesla-cap-to-humans-after-robots-undercut-model-3

[IncludeCpp] http://www.includecpp.org/

[Jordan18] Michael Jordan (2018) 'Artificial Intelligence – The Revolution Hasn't Happened Yet', https://medium.com/@mijordan3/artificial-intelligence-the-revolution-hasnt-happened-yet-5e1d5812e1e7

[Malone18] Dylan Malone (2018) https://twitter.com/dylanmalone/status/986321420761235456, posted 17 April 2018

[Musk18] Elon Musk (2018) https://twitter.com/elonmusk/statuses/984882630947753984, posted 13 April 2018

[NRC] National Research Council (2015) Reliability Growth: Enhancing Defense System Reliability. Panel on Reliability Growth Methods for Defense Systems, Committee on National Statistics, Division of Behavioral and Social Sciences and Education. Washington, DC: The National Academies Press. https://www.nap.edu/read/18987/chapter/11#122

[Procida18] Daniele Procida (2018) 'Fighting the controls: tragedy and madness for pilots and programmers', presentation at the ACCU Conference 2018: https://conference.accu.org/2018/sessions.html#XFightingthecontrolstragedyandmadnessforpilotsandprogrammers

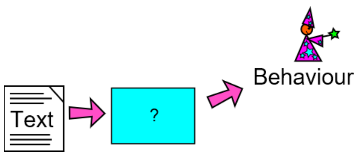
[Stackoverflow] 'Role of Bias in Neural Networks' https://stackoverflow.com/questions/2480650/role-of-bias-in-neural-networks

[Sossamon18] Jeff Sossamon (2018) 'In countries with higher gender equality, women are less likely to get STEM degrees', World Economic Forum, https://www.weforum.org/agenda/2018/02/does-gender-equality-result-in-fewer-female-stem-grads

[UNESCO] UNESCO (2018) 'Improving access to engineering careers for women in Africa and in the Arab States', http://www.unesco.org/new/en/natural-sciences/science-technology/engineering/infocus-engineering/women-and-engineering-in-africa-and-in-the-arab-states/

How to Write a Programming Language: Part 1, The Lexer

Writing a programming language might sound very difficult. Andy Balaam starts his series with a lexer.



A programming language is a program that converts text (source code) into behaviour. Because it's a program that works with other programs, it can sound complicated – even something that shouldn't be attempted by mere mortals – but really, programming languages are relatively simple programs, often much simpler than the programs they are used to write.

In this series we will be writing an interpreter for our own programming language, called Cell. Cell is a proper language, with strings and numbers, if statements, for loops, functions and things that work like objects. If you follow this series you will get a feel for what a programming language is, and be ready to start designing your own!

Most programming languages are designed to make life easy for someone: usually that person is the programmer who will be writing programs in the language. Different languages have different designs based on the needs of that person – for example, Python is designed (among other things [Peters04]) to make code easy to read, and Rust is designed (among other things [Rust]) to make it easy to avoid certain kinds of mistakes.

Cell is unusual because it is designed to make life easy for us: the people who are writing it. Lots of things about its design mean that we can write less code when we are implementing it. Sometimes, this will make life a bit more difficult for the person writing programs in Cell. We will have to live with that: Cell is a toy language, not a hardened tool.

First, let's look at an example of a program written in Cell.

A Cell program

This program shows how to make variables and call functions in Cell:

```
x = 3;
y = x + 2;
print(y);
```

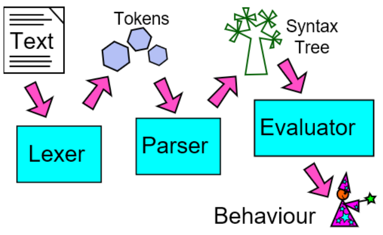
When you run it, this program will print out the value of y, which is 5.

Cell programs should be relatively familiar to people who have used a curly-brace language like C, C++ or Java, and also takes inspiration from dynamic languages like Python and Ruby. (In fact, under the covers, the language Cell looks most like is Lisp, but its syntax is different.)

The Cell interpreter we will be writing is written in Python, which was chosen because Python programs tend to be short and easy to read.

How does a programming language work?

Most programming languages are built from several parts: the lexer takes in the source code and converts it into tokens, the parser understands the structure described by the tokens and builds them into a syntax tree, and then the evaluator uses the syntax tree to decide what to do. (Of course, real languages have other parts including things like compilation, optimisation, and bytecode generation.)



Andy Balaam Andy is happy as long as he has a programming language and a problem. He finds over time he has more and more of each. You can find his open source projects at artificialworlds.net or contact him on

```
def lex(chars_iter):
  chars = PeekableStream(chars_iter)
  while chars.next is not None:
    c = chars.move_next()
    if c in " \n": pass
      # Ignore white space
    elif c in "(){},;=:": yield (c, "")
      # Special characters
    elif c in "+-*/":  yield ("operation", c)
    elif c in ("'", '"'): yield ("string",
      _scan_string(c, chars))
    elif re.match("[.0-9]", c): yield ("number",
      _scan(c, chars, "[.0-9]"))
    elif re.match("[_a-zA-Z]", c):
      yield ("symbol", _scan(c, chars,
             "[_a-zA-Z0-9]"))
    elif c == "\t": raise Exception(
      "Tabs are not allowed in Cell.")
    else: raise Exception(
      "Unexpected character: '" + c + "'.")
```

Listing 1

The Lexer

In this article we will look at the lexer – the first part of our program. The lexer takes in text (source code) and transforms it into tokens. Tokens are things like a number, a string, or a name.

In `Cell`, the types of tokens are:

- Numbers, e.g 12 or 4.2
- Strings, e.g. `"foo"` or `'bar'`
- Symbols, e.g. `baz` or `qux_Quux`
- Operators, e.g. `+` or `–`
- Special punctuation, including `(`, `}` and `;`

So, the lexer is really just a function that takes in a string (some Cell source code) and returns all the tokens it finds in that string. The main function is shown in listing 1.

The `lex` function takes in an argument called `chars_iter` that provides the characters of the code we are lexing. This can be anything that gives us single characters if we loop through it, for example an ordinary string. We immediately wrap `chars_iter` in a `PeekableStream`, which is a

```
class PeekableStream:
  def __init__(self, iterator):
    self.iterator = iter(iterator)
    self._fill()
  def _fill(self):
    try:
      self.next = next(self.iterator)
    except StopIteration:
      self.next = None
  def move_next(self):
    ret = self.next
    self._fill()
    return ret
```

Listing 2

little class (shown in listing 2) that allows us to check one character ahead in the stream of characters we are receiving.

The **lex** function returns a stream of tokens using Python's **yield** keyword to provide them one by one. Each token is a Python tuple containing two things: a type, which tells us what kind of token we are dealing with, and the value, which is the contents of the token (for example, a number, a string, or the name of a variable).

The main body of the **lex** function is a **while** loop stepping through the characters one by one, and for each one doing something based on what type of character it is. When we are in this part of the code, we know we are looking for the beginning of a new token, and the first character of that token will help us decide what type of token it is.

The first line of the **if** allows us to skip over any white space (spaces or newlines) we find. The second line identifies any special characters. The values yielded by the **lex** function are pairs that look like (TYPE, VALUE), for example (**"string", "Hello"**) would represent a string token that contains the word 'Hello'. The special characters are slightly different – we treat each character as a unique type, so when we find a **;** character, we yield (**";", ""**). This means the parser (which we will see in the next article in this series) can treat each special character differently. Because in Cell special characters are always exactly one character long, we can immediately yield a token when we find one.

The next part (**elif c in "+-*/"**) identifies arithmetic operations. Again, these tokens are always one character long, so we can immediately yield a token with type **"operation"**, and value the actual symbol the user typed.

Now we move on to more interesting tokens. If the first character of a token is a single or double quote, we know we are dealing with a string. We must scan forwards through the characters until we find a matching close quote, and then yield a token with type string. To scan forwards we call the function **_scan_string**, which is shown in listing 3.

_scan_string moves through the characters of the string, and stops when it reaches a matching quote. The characters between the quotes are returned, and this is the value of the token that is yielded from the **lex** function. Unlike most programming languages, Cell does not provide a way of 'escaping' a quote symbol by writing **\"** or similar. This is a limitation we have chosen to accept to keep our lexer simple.

```
def _scan_string(delim, chars):
  ret = ""
  while chars.next != delim:
    c = chars.move_next()
    if c is None:
      raise Exception( \
      "A string ran off the end of the program.")
    ret += c
  chars.move_next()
  return ret
```

Listing 3

```
def _scan(first_char, chars, allowed):
  ret = first_char
  p = chars.next
  while p is not None and re.match(allowed, p):
    ret += chars.move_next()
    p = chars.next
  return ret
```

Listing 4

Continuing through the big **if**/**elif** block in the **lex** function (listing 1), next we come to numbers. The first character in a number will always be a number or a decimal point, so when we see one of those we call the **_scan** function, telling it to keep consuming characters while it can see numbers or decimal points. The **_scan** function is shown in listing 4.

_scan is similar to **_scan_string**, but instead of continuing until it reaches a closing quote, it continues reading characters until it finds one that is not allowed. It uses a regular expression to handle this, and for a number, the regular expression is [.0-9], which just means only numbers and decimal points are allowed. Notice that **"0.4.3"** would count as a number here, or even **"...."**. It would certainly be nice to tell the programmer that they made a mistake like this, but we don't necessarily need to do this in the lexer – we might choose to check this in the parser, or in a later validation stage.

The next **elif** in the **lex** function (listing 1) checks for a symbol like a variable name. These must start with a letter or an underscore. Once we have found a letter or an underscore, we call the **_scan** function again, and the characters that are allowed include numbers as well as letters and underscores. Notice that at this point, the lexer does not care at all whether this is a variable name, a function name, or something else. In fact it doesn't even care whether you are allowed to write a symbol at this point in the program – all it cares about is that the characters it found make a symbol. It's the parser's job to care about what is allowed where.

It is common in lots of programming languages to allow numbers in symbols, but not for the first character. If you ever wondered why that is, this might help to explain – by disallowing numbers as the first character, we make it easy for the lexer to tell the difference between numbers and symbols, without having to scan through the whole token first.

The last two branches of the **if**/**elif** structure in the **lex** function handle tab characters (which are simply never allowed in Cell programs) and anything else that was unexpected. Both of these produce (fairly unhelpful) error messages.

We have now talked about the entire source code of Cell's lexer – that is all there is, so if you understand it, you have a good chance of understanding the lexer in your favourite programming language, or of writing your own.

You can find the whole source code for Cell at https://github.com/andybalaam/cell along with articles and videos explaining more about how it works.

Summary

Lexers do a very simple job: read in the text version of a program, and break up the parts of it into separate tokens that make sense to the next part: the parser.

Next time, we'll look at Cell's parser, and how it takes in tokens and arranges them into a tree shape reflecting the actual structure of the instructions we are giving to the computer. After that we'll look at how the evaluator turns that tree into actual behaviour, making a real, working programming language. ■

References

[Peters04] Tim Peters (2004) 'PEP 20 – The Zen of Python', https://www.python.org/dev/peps/pep-0020/, posted 19 August 2004

[Rust] *The Rust Programming Language*, available at https://doc.rust-lang.org/book/second-edition/index.html

Type-agnostic Tracing Using {fmt}

Tracing compound and custom types is a challenge. Mike Crowe demonstrates how {fmt} provides a safe alternative to printf.

About eleven years ago, I was responsible for adding types that looked a bit like Listing 1 to a young C++ code base that lacked a standard way to do tracing for debugging. The implementation of **`AlwaysTrace::operator()`** wrote the trace message to an appropriate place.

```
struct NeverTrace {
    void operator()(const char *, ...)
      __attribute__((format(printf, 2, 3))) {}
};

struct AlwaysTrace {
    void operator()(const char *, ...)
      __attribute__((format(printf, 2, 3)));
};

#if DEBUG>0
typedef AlwaysTrace DebugTrace;
#else
typedef NeverTrace DebugTrace;
#endif
```

Listing 1

The intention was that each file could declare one or more instances of these types and use it throughout the code:

```
static DebugTrace TRACE;
void my_function(int i)
{
  TRACE("A helpful message: %d\n", i);
}
```

The real classes had various other helpful functions of course, and eventually we ended up with a global **`TRACE_ERROR`** instance that would always be emitted.

This worked reasonably well, and although we always had plans to improve the mechanism, we've never really got round to it. Some adornments to enable GCC's **`printf`** format string checking and using **`-Werror`** meant that the risk of getting the format string wrong was low. It was annoying to have to use **`c_str()`** when tracing **`std::string`** instances, but we just lived with that.

I'd always planned to support tracing compound and custom types by adding some iostreams-like functionality but I never got round to it. In any case, my coworkers and I were generally not keen on iostreams for its verbosity and other reasons recently well described in [Ignatchenko18].

Mike Crowe Mike became a C++ and embedded Linux developer by accident twenty years ago and hasn't managed to escape yet. Working for small companies means that he gets to work on a wide range of high and low-level software, as well as release processes and build tools to stop him getting bored. He can be reached at accu@mcrowe.com.

```
#include <inttypes.h>
void foo(uint64_t value)
{
  TRACE("foo passed %llu\n", value);
  //...
}
void bar(size_t length)
{
  TRACE("bar passed %zu\n", length);
  //...
}
```

Listing 2

We continued to use these tracing classes for many years whilst targeting 32-bit MIPS and later ARM targets. We ended up with code like Listing 2.

This all worked fine, until we tried to compile for our first 64-bit target.

The ugliness of <inttypes.h>

The **`<inttypes.h>`** header (along with the **`<cinttypes>`** that we probably ought to have been using from C++) contains handy definitions for types with specific numbers of bits. It contains declarations like:

```
#if defined(__LP64)
typedef unsigned long uint64_t;
typedef long int64_t;
#else
typedef unsigned long long uint64_t;
typedef long long int64_t;
#endif
```

Unfortunately, this means that when compiling for a 32-bit target the **`%llu`** and **`%lld`** format specifiers are correct, but when compiling for a 64-bit target **`%lu`** and **`%ld`** must be used. **`<inttypes.h>`** provides the **`PRIu64`** and **`PRId64`** macros that expand to the correct form. Unfortunately, since they need to be used in a string literal, they are extremely ugly to use. (The full set of macros is at [CppRef].)

The relatively sensible:

```
void ReportProgress(uint64_t current,
  uint64_t max)
{
  TRACE("Progress %llu/%llu %llu%%\n", current,
    max, current * 100 / max);
}
```

becomes the hard-to-read, hard-to-type and hard-to-maintain:

```
void ReportProgress(uint64_t current,
  uint64_t max)
{
  TRACE("Progress %" PRIu64 "/" PRIu64 " " PRIu64
    "%%\n", current, max, current * 100 / max);
}
```

So I went looking for an alternative and I found {fmt}.

for me, the most interesting feature was its **printf-compatible interface** ... the **solution to my 64-bit tracing problem** without needing to change all my code

{fmt}

The {fmt} library [fmt] provides a safe alternative to `printf`. It provides a Python-like formatting mechanism for C++, taking advantage of C++ type safety:

```
fmt::print("The {} is {}{}\n", "answer", 42L, '.');
```

But for me, the most interesting feature was its `printf`-compatible interface. When using this interface it will parse `printf` format strings, but mostly disregard the type information. This means that you can write:

```
void ReportProgress(uint64_t current,
 uint64_t max)
{
  TRACE("Progress %u/%u %u%%\n", current, max,
    current * 100 / max);
}
```

without caring whether the three integers are `unsigned long` or `unsigned long long`, or any other integer type for that matter. This turned out to be the solution to my 64-bit tracing problem without needing to change all my code.

Using fmt::printf

The first step was to implement `AlwaysTrace::operator()` in terms of `fmt::printf` (see Listing 3).

Catching exceptions

This worked, but if I accidentally wrote:

```
void ReportProgress(uint64_t current,
  uint64_t max)
{
  TRACE("Progress %d/%d%%\n", current);
}
```

then GCC's format string checker was no longer there to generate a compilation error; {fmt} would instead throw a `FormatError` exception. If there are too many parameters then it silently ignores the extra ones.

```
struct NeverTrace {
  template <typename... Args>
  void operator()(const char *format,
    Args&&... args) {}
};
struct AlwaysTrace {
  template <typename... Args>
  void operator()(const char *format,
    Args&&... Args) {
      fmt::printf(format,
        std::forward<Args>(args)...);
    }
};
```

Listing 3

```
namespace wrapped {
  template <typename... Args>
  inline void printf(const char *format,
      Args&&... args) {
    try {
      fmt::printf(format,
        std::forward<Args>(args)...);
    }
    catch (const std::exception &e) {
      fmt::printf("Message '%s' threw '%s'\n",
        format, e.what());
    }
  }
}
struct AlwaysTrace {
  template <typename... Args>
  void operator()(const char *format,
    Args&&... Args) {
      wrapped::printf(format,
        std::forward<Args>(args)...);
    }
};
```

Listing 4

Tracing isn't important enough for us to want to risk terminating the program. This is especially important if tracing is not always enabled. I considered modifying {fmt} itself to stop it throwing, but that would mean disabling exception throwing even if we were to make use of the library in normal code too.

The simplest solution is to just catch exceptions and emit an error message along with the format string so that the offending line can be found and fixed (see Listing 4).

Just before this article went to print, version 5 of {fmt} was released. This version supports compile-time verification of Python-style format strings [Zverovich17]. I look forward to being able to take advantage of this to support both the Python-style format strings and validation whilst keeping the existing `printf`-compatible functions for existing code.

Tracing errors

In many places we'd taken advantage of glibc's `%m` format specifier to print the error message associated with the current value of `errno`. The upstream maintainer didn't want to support such an extension [Crowe17a], so I applied a local patch to do so. I hope to come up with something similar that will be acceptable upstream in the future.

Tracing pointers

In certain sections of the code there were a large number of places where the `this` pointer was emitted as part of a trace message. This helped to tie the tracing from multiple instances together when reading the logs. It used the `%p` format specifier for this.

Now that we're using the library, we no longer have to care about the exact types of integers we're tracing

The {fmt} library supports the **%p** format specifier, but the corresponding parameter must be of **const void *** or **void *** type. It has good reasons for this, at least some of which are described in [Wilson09]. On its branch hopefully destined for standardisation, it provides a **fmt::ptr**` helper to cast arbitrary pointers to an acceptable type.

Unfortunately, I had lots of code using **%p** and casting all the corresponding parameters was painful. I decided to patch {fmt} locally to support arbitrary pointers [Crowe17b], despite the downsides. The upstream maintainer seems favourable to supporting this in the **printf**-compatible case only, if only I can find a clean way to do so.

Effect on compilation time and binary size

Of course, all this magic has to come at a cost. In my not-hugely-scientific experiments, on a project that contained about 7500 trace statements, compilation and linking with GCC of debug builds (with **-g3**) took about 5% longer. The same with **-O2** and **-g3** only took about 1% longer. So, whilst there is an effect, it's not huge.

This code runs on an embedded system, albeit one with over a gigabyte of memory, but nonetheless the size of the generated code was a concern. Without making any attempt to catch exceptions the stripped binary was about 0.75% bigger. When catching exceptions it was about 1.5% bigger.

Enabling new behaviours

Now that we're using the library, we no longer have to care about the exact types of integers we're tracing, and can pass **std::string** without calling **c_str()**, which is a bonus. We no longer have to be picky about the exact type of integers either. In new code we don't have to remember to say **%zu** when tracing a **size_t**.

Tracing custom types

We end up tracing **std::chrono** types in quite a few places. We can define a custom formatter with something like Listing 5, then write code like:

```
void f() {
  auto start = std::chrono::steady_clock::now();
  do_something_time_consuming();
  auto duration =
    std::chrono::steady_clock::now() - start;
  TRACE("It took %ldms\n", duration);
}
```

The future

Victor Zverovich, the primary author of the {fmt} library, has proposed the Python-style format strings subset of the library for standardisation and presented his work at the Jacksonville WG21 meeting [WG21]. I'm keen to see how it can be used to improve our tracing in the future. ■

Thanks

Thanks to Victor Zverovich, the primary author of {fmt}, for reviewing draft versions of this article and for helping with my attempts to mould it to our needs.

Thanks to Frances Buontempo and the *Overload* reviewers for their suggestions and advice.

References

[CppRef] 'Fixed width integer types' available at: http://en.cppreference.com/w/cpp/types/integer

[Crowe17a] Mike Crowe (2017) 'printf: Add support for glibc's %m format' at https://github.com/fmtlib/fmt/pull/550, posted 22 July 2017

[Crowe17b] Mike Crowe (2017) 'What are the downsides to disabling the private MakeValue::MakeValue<T*> overloads?' at https://github.com/fmtlib/fmt/issues/608, posted 11 November 2017

[fmt] The {fmt} library http://fmtlib.net

[Ignatchenko18] Sergey Ignatchenko (2018) 'This is NOT yet another printf-vs-cout debate' in *Overload* 144, April 2018, available at: https://accu.org/index.php/journals/2486

[WG21] 'Text Formatting at the ISO C++ standards meeting in Jacksonville', available online at http://www.zverovich.net/2018/03/17/text-formatting-jacksonville.html

[Wilson09] Matthew Wilson (2009) 'An Introduction to Fast Format (Part 1): The State of the Art', *Overload* 89, available at https://accu.org/index.php/journals/1539

[Zverovich17] Victor Zverovich (2017) Verification of Python-style format strings: http://zverovich.net/2017/11/05/compile-time-format-strings.html

```
namespace fmt {
    /// Displays in microseconds or seconds
    template <typename Rep, typename Period>
    void format_arg(fmt::BasicFormatter<char> &f,
      const char *&format_str,
      const std::chrono::duration<Rep, Period>
        &duration)
      {
        if (duration < std::chrono::seconds(1))
          f.writer().write("{}us",
          std::chrono::duration<double,
          std::micro>(duration).count());
        else
          f.writer().write("{}s",
          std::chrono::duration<double>(duration)
            .count());
      }
}
```

Listing 5

A Short Overview of Object Oriented Software Design

Object oriented design has many principles. Stanislav Kozlovski demonstrates good design through a role playing game.

Most modern programming languages support and encourage object-oriented programming (OOP). Even though lately we seem to be seeing a slight shift away from this, as people start using languages which are not *heavily* influenced by OOP (such as Go, Rust, Elixir, Elm, Scala), most still have objects. The design principles we are going to outline here apply to non-OOP languages as well.

To succeed in writing clear, high-quality, maintainable and extendable code you will need to know about design principles that have proven themselves effective over decades of experience.

Disclosure: The example we are going to be going through will be in Python. Examples are there to prove a point and may be sloppy in other, obvious, ways.

Object types

Since we are going to be modelling our code around objects, it would be useful to differentiate between their different responsibilities and variations.

There are three type of objects:

1. Entity object

This object generally corresponds to some real-world entity in the problem space. Say we're building a role-playing game (RPG), an entity object would be our simple **`Hero`** class (Listing 1).

These objects generally contain properties about themselves (such as **`health`** or **`mana`**) and are modifiable through certain rules.

2. Control object

Control objects (sometimes also called *Manager objects*) are responsible for the coordination of other objects. These are objects that *control* and make use of other objects. A great example in our RPG analogy would be the **`Fight`** class, which controls two heroes and makes them fight (Listing 2).

```
class Hero:
    def __init__(self, health, mana):
        self._health = health
        self._mana = mana

    def attack(self) -> int:
        """
        Returns the attack damage of the Hero
        """
        return 1

    def take_damage(self, damage: int):
        self._health -= damage

    def is_alive(self):
        return self._health > 0
```

Listing 1

```
class Fight:
    class FightOver(Exception):
        def __init__(self, winner, *args,
                     **kwargs):
            self.winner = winner
            super(*args, **kwargs)

    def __init__(self, hero_a: Hero,
                 hero_b: Hero):
        self._hero_a = hero_a
        self._hero_b = hero_b
        self.fight_ongoing = True
        self.winner = None

    def fight(self):
        while self.fight_ongoing:
            self._run_round()
        print(
            'The fight has ended! Winner is #{}'.\
            format(self.winner))

    def _run_round(self):
        try:
            self._run_attack(self._hero_a,
                             self._hero_b)
            self._run_attack(self._hero_b,
                             self._hero_a)
        except self.FightOver as e:
            self._finish_round(e.winner)

    def _run_attack(self, attacker: Hero,
                    victim: Hero):
        damage = attacker.attack()
        victim.take_damage(damage)
        if not victim.is_alive():
            raise self.FightOver(winner=attacker)

    def _finish_round(self, winner: Hero):
        self.winner = winner
        self.fight_ongoing = False
```

Listing 2

Stanislav Kozlovski has been programming since the age of 19 – and is now 21. He spent a year racing through some coding academies and bootcamps, where he aced all of his courses, and took a job at a Berlin company aiming to become the first ever global card acceptance brand. Contact him on github (where he's enether) or at Stanislav_Kozlovski@outlook.com

Encapsulating the logic for a fight in such a class provides you with multiple benefits: one of which is the easy extensibility of the action. You can very easily pass in a non-player character (NPC) type for the hero to fight, provided it exposes the same API. You can also very easily inherit the class and override some of the functionality to meet your needs.

3. Boundary object

These are objects which sit at the boundary of your system. Any object which takes input from or produces output to another system – regardless if that system is a User, the internet or a database – can be classified as a boundary object (Listing 3).

These boundary objects are responsible for translating information into and out of our system. In an example where we take User commands, we would need the boundary object to translate a keyboard input (like a spacebar) into a recognizable domain event (such as a character jump).

Bonus: Value object

Value objects [Wikipedia-1] represent a simple value in your domain. They are immutable and have no identity.

If we were to incorporate them into our game, a `Money` or `Damage` class would be a great fit. Said objects let us easily distinguish, find and debug related functionality, while the naive approach of using a primitive type – an array of integers or one integer– does not (Listing 4).

They can be classified as a subcategory of `Entity` objects.

Key design principles

Design principles are rules in software design that have proven themselves valuable over the years. Following them strictly will help you ensure your software is of top-notch quality.

```
class UserInput:
    def __init__(self, input_parser):
        self.input_parser = input_parser

    def take_command(self):
        """
        Takes the user's input,
    parses it into a recognizable command
    and returns it
        """
        command = self._parse_input(
            self._take_input())
        return command

    def _parse_input(self, input):
        return self.input_parser.parse(input)

    def _take_input(self):
        raise NotImplementedError()


class UserMouseInput(UserInput):
    pass


class UserKeyboardInput(UserInput):
    pass


class UserJoystickInput(UserInput):
    pass
```

Listing 3

```
class Money:
    def __init__(self, gold, silver, copper):
        self.gold = gold
        self.silver = silver
        self.copper = copper

    def __eq__(self, other):
        return self.gold == other.gold and \
          self.silver == other.silver and \
          self.copper == other.copper

    def __gt__(self, other):
        if self.gold == other.gold and \
          self.silver == other.silver:
            return self.copper > other.copper
        if self.gold == other.gold:
            return self.silver > other.silver

        return self.gold > other.gold

    def __add__(self, other):
        return Money(
            gold=self.gold + other.gold,
            silver=self.silver + other.silver,
            copper=self.copper + other.copper)

    def __str__(self):
        return 'Money Object(' + \
           'Gold: {}; Silver: {}; Copper: {})'.\
           format(self.gold,
                self.silver,
                self.copper)

    def __repr__(self):
        return self.__str__()


print(Money(1, 1, 1) == Money(1, 1, 1))
# => True
print(Money(1, 1, 1) > Money(1, 2, 1))
# => False
print(Money(1, 1, 0) + Money(1, 1, 1))
# => Money Object(Gold: 2; Silver: 2; Copper: 1)
```

Listing 4

Abstraction

Abstraction is the idea of simplifying a concept to its bare essentials in some context. It allows you to better understand the concept by stripping it down to a simplified version.

The examples above illustrate abstraction – look at how the `Fight` class is structured. The way you use it is as simple as possible – you give it two heroes as arguments in instantiation and call the `fight()` method. Nothing more, nothing less.

Abstraction in your code should follow the rule of least surprise [Wikipedia-2]. Your abstraction should not surprise anybody with needless and unrelated behavior/properties. In other words – it should be intuitive.

Note that our `Hero#take_damage()` function does not do something unexpected, like delete our character upon death. But we can expect it to kill our character if his health goes below zero.

Encapsulation

Encapsulation can be thought of as putting something inside a capsule – you limit its exposure to the outside world. In software, restricting access to inner objects and properties helps with data integrity.

Encapsulation black-boxes inner logic and makes your classes easier to manage, because you know what part is used by other systems and what isn't. This means that you can easily rework the inner logic while retaining the public parts and be sure that you have not broken anything.

```
class Hero:
    def __init__(self, health, mana):
        self._health = health
        self._mana = mana
        self._strength = 0
        self._agility = 0
        self._stamina = 0
        self.level = 0
        self._items = {}
        self._equipment = {}
        self._item_capacity = 30
        self.stamina_buff = None
        self.agility_buff = None
        self.strength_buff = None
        self.buff_duration = -1

    def level_up(self):
        self.level += 1
        self._stamina += 1
        self._agility += 1
        self._strength += 1
        self._health += 5

    def take_buff(self, stamina_increase,
                  strength_increase,
                  agility_increase):
        self.stamina_buff = stamina_increase
        self.agility_buff = agility_increase
        self.strength_buff = strength_increase
        self._stamina += stamina_increase
        self._strength += strength_increase
        self._agility += agility_increase
        self.buff_duration = 10  # rounds

    def pass_round(self):
        if self.buff_duration > 0:
            self.buff_duration -= 1
        if self.buff_duration == 0:  # Remove buff
            self._stamina -= self.stamina_buff
            self._strength -= self.strength_buff
            self._agility -= self.agility_buff
            self._health -= self.stamina_buff * 5
            self.buff_duration = -1
            self.stamina_buff = None
            self.agility_buff = None
            self.strength_buff = None

    def attack(self) -> int:
        """
        Returns the attack damage of the Hero
        """
        return 1 + (self._agility * 0.2) + (
            self._strength * 0.2)

    def take_damage(self, damage: int):
        self._health -= damage

    def is_alive(self):
        return self._health > 0

    def take_item(self, item: Item):
        if self._item_capacity == 0:
            raise Exception('No more free slots')
        self._items[item.id] = item
        self._item_capacity -= 1
```

Listing 5

As a side-effect, working with the encapsulated functionality from the outside becomes simpler as you have less things to think about.

In most languages, this is done through the so-called access modifiers (private, protected, and so on) [Wikipedia-3]. Python is not the best example of this, as it lacks such explicit modifiers built into the runtime, but we use conventions to work around this. The **_** prefix to the variables/methods denote them as being private.

For example, imagine we change our **Fight#_run_attack** method to return a boolean variable that indicates if the fight is over rather than raise an exception. We will know that the only code we might have broken is inside the **Fight** class, because we made the method private.

Remember, code is more frequently changed than written anew. Being able to change your code with as clear and little repercussions as possible is flexibility you want as a developer.

Decomposition

Decomposition is the action of splitting an object into multiple separate smaller parts. Said parts are easier to understand, maintain and program.

Imagine we wanted to incorporate more RPG features like buffs, inventory, equipment and character attributes on top of our **Hero** (see Listing 5).

I assume you can tell this code is becoming pretty messy. Our **Hero** object is doing too much stuff at once and this code is becoming pretty brittle as a result of that.

For example, one stamina point is worth 5 health. If we ever want to change this in the future to make it worth 6 health, we'd need to change the implementation in multiple places.

The answer is to decompose the **Hero** object into multiple smaller objects which each encompass some of the functionality (Figure 1, overleaf, and Listing 6).

Now, after decomposing our Hero object's functionality into **HeroAttributes**, **HeroInventory**, **HeroEquipment** and **HeroBuff** objects, adding future functionality will be easier, more encapsulated and better abstracted. You can tell our code is way cleaner and clearer on what it does.

There are three types of decomposition relationships:

- **association**: Defines a loose relationship between two components. Both components do not depend on one another but may work together.

  Example: **Hero** and a **Zone** object.
- **aggregation**: Defines a weak 'has-a' relationship between a whole and its parts. Considered weak, because the parts can exist without the whole.

  Example: **HeroInventory** and **Item**.

  A **HeroInventory** can have many **Items** and an **Item** can belong to any **HeroInventory** (such as trading items).
- **composition**: A strong 'has-a' relationship where the whole and the part cannot exist without each other. The parts cannot be shared, as the whole depends on those exact parts.

  Example: **Hero** and **HeroAttributes**.

  These are the Hero's attributes – you cannot change their owner.

```
    def equip_item(self, item: Item):
        if item.id not in self._items:
            raise Exception(
                'Item is not present in inventory!'
            )
        self._equipment[item.slot] = item
        self._agility += item.agility
        self._stamina += item.stamina
        self._strength += item.strength
        self._health += item.stamina * 5
```

Listing 5 (cont'd)

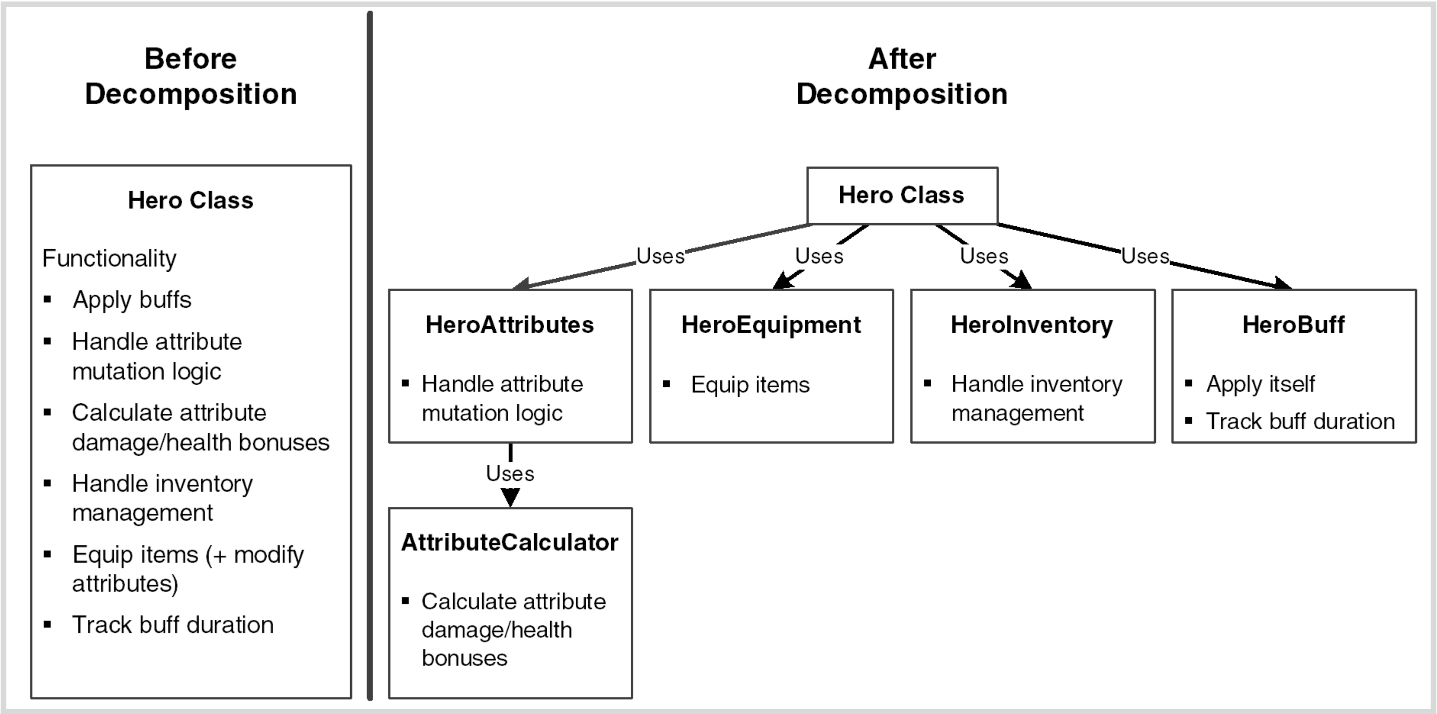

Before Decomposition
Hero Class
Functionality
Apply buffs
Handle attribute mutation logic
Calculate attribute damage/health bonuses
Handle inventory management
Equip items (+ modify attributes)
Track buff duration
After Decomposition
Hero Class
Uses
Uses
Uses
Uses
HeroAttributes
Handle attribute mutation logic
HeroEquipment
Equip items
HeroInventory
Handle inventory management
HeroBuff
Apply itself
Track buff duration
Uses
AttributeCalculator
Calculate attribute damage/health bonuses

Figure 1

Generalization

Generalization might be the most important design principle – it is the process of extracting shared characteristics and combining them in one place. All of us know about the concept of functions and class inheritance – both are a kind of generalization.

A comparison might clear things up: while *abstraction* reduces complexity by hiding unnecessary detail, *generalization* reduces

```
from copy import deepcopy


class AttributeCalculator:
    @staticmethod
    def stamina_to_health(self, stamina):
        return stamina * 6

    @staticmethod
    def agility_to_damage(self, agility):
        return agility * 0.2

    @staticmethod
    def strength_to_damage(self, strength):
        return strength * 0.2


class HeroInventory:
    class FullInventoryException(Exception):
        pass

    def __init__(self, capacity):
        self._equipment = {}
        self._item_capacity = capacity

    def store_item(self, item: Item):
        if self._item_capacity < 0:
            raise self.FullInventoryException()
        self._equipment[item.id] = item
        self._item_capacity -= 1

    def has_item(self, item):
        return item.id in self._equipment
```

Listing 6

```
class HeroAttributes:
    def __init__(self, health, mana):
        self.health = health
        self.mana = mana
        self.stamina = 0
        self.strength = 0
        self.agility = 0
        self.damage = 1

    def increase(self,
                 stamina=0,
                 agility=0,
                 strength=0):
        self.stamina += stamina
        self.health += \
        AttributeCalculator.stamina_to_health(
           stamina)
        self.damage += \
        AttributeCalculator.strength_to_damage(
            strength
        ) + AttributeCalculator.agility_to_damage(
            agility)
        self.agility += agility
        self.strength += strength

    def decrease(self,
                 stamina=0,
                 agility=0,
                 strength=0):
        self.stamina -= stamina
        self.health -= \
        AttributeCalculator.stamina_to_health(
            stamina)
        self.damage -= \
        AttributeCalculator.strength_to_damage(
            strength
        ) + AttributeCalculator.agility_to_damage(
            agility)
        self.agility -= agility
        self.strength -= strength
```

Listing 6 (cont'd)

```
class HeroEquipment:
    def __init__(self,
                 hero_attributes: HeroAttributes):
        self.hero_attributes = hero_attributes
        self._equipment = {}

    def equip_item(self, item):
        self._equipment[item.slot] = item
        self.hero_attributes.increase(
            stamina=item.stamina,
            strength=item.strength,
            agility=item.agility)


class HeroBuff:
    class Expired(Exception):
        pass

    def __init__(self, stamina, strength, agility,
                 round_duration):
        self.attributes = None
        self.stamina = stamina
        self.strength = strength
        self.agility = agility
        self.duration = round_duration

    def with_attributes(
            self,
            hero_attributes: HeroAttributes):
        buff = deepcopy(self)
        buff.attributes = hero_attributes
        return buff

    def apply(self):
        if self.attributes is None:
            raise Exception()
        self.attributes.increase(
            stamina=self.stamina,
            strength=self.strength,
            agility=self.agility)

    def deapply(self):
        self.attributes.decrease(
            stamina=self.stamina,
            strength=self.strength,
            agility=self.agility)

    def pass_round(self):
        self.duration -= 0
        if self.has_expired():
            self.deapply()
            raise self.Expired()

    def has_expired(self):
        return self.duration == 0
```

Listing 6 (cont'd)

```
class Hero:
    def __init__(self, health, mana):
        self.attributes = HeroAttributes(
            health, mana)
        self.level = 0
        self.inventory = HeroInventory(
            capacity=30)
        self.equipment = HeroEquipment(
            self.attributes)
        self.buff = None

    def level_up(self):
        self.level += 1
        self.attributes.increase(1, 1, 1)

    def attack(self) -> int:
        """
        Returns the attack damage of the Hero
        """
        return self.attributes.damage

    def take_damage(self, damage: int):
        self.attributes.health -= damage

    def take_buff(self, buff: HeroBuff):
        self.buff = buff.with_attributes(
            self.attributes)
        self.buff.apply()

    def pass_round(self):
        if self.buff:
            try:
                self.buff.pass_round()
            except HeroBuff.Expired:
                self.buff = None

    def is_alive(self):
        return self.attributes.health > 0

    def take_item(self, item: Item):
        self.inventory.store_item(item)

    def equip_item(self, item: Item):
        if not self.inventory.has_item(item):
            raise Exception(
               'Item is not present in inventory!'
            )
        self.equipment.equip_item(item)
```

Listing 6 (cont'd)

complexity by replacing multiple entities which perform similar functions with a single construct (Listing 7).

In the given example, we have generalized our common **Hero** and **NPC** classes' functionality into a common ancestor called **Entity**. This is always achieved through inheritance.

Here, instead of having our **NPC** and **Hero** classes implement all the methods twice and violate the DRY principle [Wikipedia-4], we reduced the complexity by moving their common functionality into a base class.

As a forewarning – do not overdo inheritance. Many experienced people recommend you favor composition over inheritance [StackExchange] [Stackoverflow] [Wikipedia-5].

Inheritance is often abused by amateur programmers, probably because it is one of the first OOP techniques they grasp due to its simplicity.

Composition

Composition is the principle of combining multiple objects into a more complex one. Practically said – it is creating instances of objects and using their functionality instead of directly inheriting it.

An object that uses composition can be called a *composite object*. It is important that this composite is simpler than the sum of its peers. When combining multiple classes into one we want to raise the level of abstraction higher and make the object simpler.

The composite object's API [Gazarov16] must hide its inner components and the interactions between them. Think of a mechanical clock, it has three hands for showing the time and one knob for setting – but internally contains dozens of moving and inter-dependent parts.

As I said, composition is preferred over inheritance, which means you should strive to move common functionality into a separate object which classes then use – rather than stash it in a base class you've inherited.

```
# Two methods which share common characteristics
def take_physical_damage(self, physical_damage):
    print('Took {} physical damage'.format(
        physical_damage))
    self._health -= physical_damage


def take_spell_damage(self, spell_damage):
    print('Took {} spell damage'.format(
        spell_damage))
    self._health -= spell_damage


# vs.


# One generalized method
def take_damage(self, damage, is_physical=True):
    damage_type = 'physical' if is_physical \
        else 'spell'
    print('Took {} {damage_type} damage'.format(
        damage))
    self._health -= damage


class Entity:
    def __init__(self):
        raise Exception(
            'Should not be initialized directly!')

    def attack(self) -> int:
        """
        Returns the attack damage of the Hero
        """
        return self.attributes.damage

    def take_damage(self, damage: int):
        self.attributes.health -= damage

    def is_alive(self):
        return self.attributes.health > 0


class Hero(Entity):
    pass


class NPC(Entity):
    pass
```

Listing 7

Let's illustrate a possible problem with over-inheriting functionality:

We just added movement to our game (Listing 8).

As we learned, instead of duplicating the code we used generalization to put the `move_right` and `move_left` functions into the `Entity` class.

Okay, now what if we wanted to introduce mounts into the game?

Figure 2 shows a good mount :)

Mounts would also need to move left and right but do not have the ability to attack. Come to think of it – they might not even have health!

I know what your solution is:

Simply move the move logic into a separate `MoveableEntity` or `MoveableObject` class which only has that functionality. The `Mount` class can then inherit that.

Then, what do we do if we want mounts that have health but cannot attack? More splitting up into subclasses? I hope you can see how our class hierarchy would begin to become complex even though our business logic is still pretty simple.

```
class Entity:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        raise Exception(
            'Should not be initialized directly!')

    def attack(self) -> int:
        """
        Returns the attack damage of the Hero
        """
        return self.attributes.damage

    def take_damage(self, damage: int):
        self.attributes.health -= damage

    def is_alive(self):
        return self.attributes.health > 0

    def move_left(self):
        self.x -= 1

    def move_right(self):
        self.x += 1


class Hero(Entity):
    pass


class NPC(Entity):
    pass
```

Listing 8

A somewhat better approach would be to abstract the movement logic into a `Movement` class (or some better name) and instantiate it in the classes which might need it. This will nicely package up the functionality and make it reusable across all sorts of objects not limited to `Entity`.

Hooray, composition!

Critical thinking disclaimer

Even though these design principles have been formed through decades of experience, it is still extremely important that you are able to think critically before blindly applying a principle to your code.

Like all things, too much can be a bad thing. Sometimes principles can be taken too far, you can get too clever with them and end up with something that is actually harder to work with.

Figure 2

As an engineer, your main trait is to critically evaluate the best approach for your unique situation, not blindly follow and apply arbitrary rules.

Cohesion, coupling and separation of concerns

Cohesion

Cohesion represents the clarity of responsibilities within a module or in other words – its complexity.

If your class performs one task and nothing else, or has a clear purpose – that class has high cohesion. On the other hand, if it is somewhat unclear in what it's doing or has more than one purpose – it has low cohesion.

You want your classes to have high cohesion. They should have only one responsibility and if you catch them having more – it might be time to split it.

Coupling

Coupling captures the complexity between connecting different classes. You want your classes to have as little and as simple connections to other classes as possible, so that you can swap them out in future events (like changing web frameworks). The goal is to have *loose coupling*.

In many languages this is achieved by heavy use of interfaces – they abstract away the specific class handling the logic and represent a sort of adapter layer in which any class can plug itself in.

Separation of Concerns

Separation of Concerns (SoC) is the idea that a software system must be split into parts that do not overlap in functionality. Or, as the name says, each 'concern' – a general term about anything that provides a solution to a problem – must be separated from the others and handled in different places.

A web page is a good example of this – it has its three layers (Information, Presentation and Behavior) separated into three places (HTML, CSS and JavaScript respectively) [Pocklington13].

If you look again at the RPG Hero example, you will see that it had many concerns at the very beginning (apply buffs, calculate attack damage, handle inventory, equip items, manage attributes). We separated those concerns through *decomposition* into more *cohesive* classes which *abstract* and *encapsulate* their details. Our `Hero` class now acts as a composite object and is much simpler than before.

Payoff

Applying such principles might look overly complicated for such a small piece of code. The truth is it a must for any software project that you plan to develop and maintain in the future. Writing such code has a bit of overhead at the very start but pays off multiple times in the long run.

These principles ensure our system is more:

- **Extendable:** *High cohesion* makes it easier to implement new modules without concern of unrelated functionality. *Low coupling* means that a new module has less stuff to connect to therefore it is easier to implement.
- **Maintainable:** *Low coupling* ensures a change in one module will generally not affect others. *High cohesion* ensures a change in system requirements will require modifying as little number of classes as possible.
- **Reusable:** *High cohesion* ensures a module's functionality is complete and well-defined. *Low coupling* makes the module less dependent on the rest of the system, making it easier to reuse in other software.

The restriction imposed by the physical column width in a printed journal means that we have had to wrap some of the lines of code in 'unusual' ways. We apologise to Stanislav for ruining his code layout, and to anyone else who finds this makes it more difficult to follow the examples.

Summary

We started off by introducing some basic high-level object types (Entity, Boundary and Control).

We then learned key principles in structuring said objects (Abstraction, Generalization, Composition, Decomposition and Encapsulation).

To follow up we introduced two software quality metrics (Coupling and Cohesion) and learned about the benefits of applying said principles.

I hope this article provided a helpful overview of some design principles. If you wish to further educate yourself in this area, here are some resources I would recommend. ■

Further reading

Design Patterns: Elements of Reusable Object-Oriented Software – Arguably the most influential book in the field. A bit dated in its examples (C++ 98) but the patterns and ideas remain very relevant.

Growing Object-Oriented Software Guided by Tests – A great book which shows how to practically apply principles outlined in this article (and more) by working through a project.

Effective Software Design – A top notch blog containing much more than design insights.

Software Design and Architecture Specialization – A great series of 4 video courses which teach you effective design throughout its application on a project that spans all four courses.

References

[Gazarov16] Petr Gazarov (2016) 'What is an API? In English, please.', https://medium.freecodecamp.org/what-is-an-api-in-english-please-b880a3214a82, posted 13 August 2016

[Pocklington13] Rob Pocklington (2013), 'Respect the Javascript', https://shinesolutions.com/2013/10/29/respect-the-javascript/, posted 29 October 2013

[StackExchange] 'Why is inheritance generally viewed as a bad thing by OOP proponents', https://softwareengineering.stackexchange.com/questions/260343/why-is-inheritance-generally-viewed-as-a-bad-thing-by-oop-proponents

[Stackoverflow] 'Prefer composition over inheritance?', https://stackoverflow.com/questions/49002/prefer-composition-over-inheritance/53354#53354

[Wikipedia-1] 'Value object', https://en.wikipedia.org/wiki/Value_object

[Wikipedia-2] 'Principle of least astonishment', https://en.wikipedia.org/wiki/Principle_of_least_astonishment

[Wikipedia-3] 'Access modifiers', https://en.wikipedia.org/wiki/Access_modifiers

[Wikipedia-4] 'Don't repeat yourself', https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

[Wikipedia-5] 'Design Patterns', https://en.wikipedia.org/wiki/Design_Patterns#Introduction,_Chapter_1

Afterwood

The curse of Agile. Chris Oldwood outlines how it affects his day at work.

The Buggles once informed us that video killed the radio star. Maybe that was true back in the late 1970s but today it's 'Agile' that's killing her instead. Time was when you could nestle into your office chair, slip on a monster pair of noise-cancelling headphones (the ones that made you look like a Cyberman) and then really get deeply embroiled in a spot of coding whilst being surrounded by your favourite tunes! There was no chance of any interruptions either as long as you didn't open your email client.

Not anymore, not now that we're all Agile. Such anti-social behaviour is considered verboten in this new world of total collaboration and 'value driven development'. There is no room for music and isolation in a team which thrives on pairs or mobs of developers huddling around a single machine all focused on the singular problem of highest worth. Deodorant is an absolute essential throughout the day and any thoughts of eating garlic for lunch will be met with an afternoon of faux hand-wafting gestures and nose pegs.

If you're partial to a nice large lunch and the occasional cheeky pint or two in the hopes that the afternoon can largely drift past as you drag out the low hanging fruit which you saved for such a special occasion, think again. The afternoon continues on from where the morning left off, albeit in a different huddle of people embracing the change in dynamic that keeps the energy flowing all throughout the day until knocking off time. If 5pm comes and you've not already left – exhausted from hours of collaborative designing, coding and testing – then your 'value generation' meter can't be full.

Energised work is the name of the game. It's not just lunchtime drinking that you'll knock on the head but a solid evening out drinking in the pub seems great at the time but the following morning when you've got a stinking hangover and you're looking to just muddle through the day you can forget it. The rest of your team will have had their full quota of 8 hours sleep and be buzzing, ready to push another handful of user stories through the pipeline and out to your loyal band of customers. Time waits for no man, and neither does the release train either.

Good old fashioned tasks used to take weeks or months, not days or even just a few hours! If you had to estimate, you had so much leeway you could easily factor in some extra hours over-and-above any genuine contingency to include personal administration time as well. We may have a day job but we also need car and home insurance, there are holidays to book, and who risks appearing absent by going out to an actual physical supermarket when you can shop online and appear to be working at the same time? If you only ever learned one keyboard shortcut I can almost guarantee it was Alt+Tab – the 'boss' key-combo.

While it's no longer possible to pad out the estimate for your latest feature to include time sorting out this summer's touring holiday around Europe, you'd still hope to squeeze in the odd household chore here and there. But no, it's not just you giving estimates these days; the whole team gets to add their tuppence worth in a planning poker session. Assuming you can even engineer being the (only) one to implement any particular feature, any attempt to game the system by giving an overly pessimistic estimate will be met with a public inquiry from your teammates to understand why your opinion sits out on the periphery of the Normal curve. Sprint planning is a well-intentioned exercise but it makes holiday planning an absolute nightmare.

Everything has to be transparent nowadays; the stakeholders get to see where their money is being spent and they're also allowed to decide on a daily basis whether it's still going in the direction they want it to. The morning stand-up is just another example of the do-gooders getting to put their oar in and disrupt the traditional art of procrastination. If you failed at the planning session to broaden the timescales enough then don't expect to simply eat into any contingency during implementation without anyone noticing. You can almost guarantee some busybody will spot at the stand-up that you're 'stuck' and will then play the 'swarming' card by sacrificing their own opportunity to arrange an appointment with the dentist by getting your 'more valuable' work back on track instead.

We now live in a world of software development that refuses to tolerate any waste; our entire process is predicated on trimming the fat to keep the team lean and nimble. Your pairing or mobbing buddies won't even let you put a single keystroke out of place without calling it out, let along managing to slip your own pet feature in under the radar. Instead of a metaphorical Jiminy Cricket [Wikipedia] keeping you on the straight and narrow you've now got real ones. The codebase is a meritocracy and every single line has to earn its keep.

None of this should be a surprise, though. Back in the early 70s, long before The Buggles even existed, Gerry Weinberg published his seminal book about the psychology of computer programming [Weinberg71] in which he identified the social nature of programming and how teams need to have collaboration as a cornerstone of their behaviour.

Yes, I'm sad that my fancy headphones mostly lay dormant in my bag along with a stack of CDs from Christmas and birthdays that I've still yet to find time to listen to. My desk is largely empty and chair configuration is far from optimal but I spend so much time at the whiteboard and at other people's desks that it hardly seems to matter. Yes, I could shave valuable time off my morning routine by short-circuiting my personal hygiene if I didn't have to share office space with my colleagues.

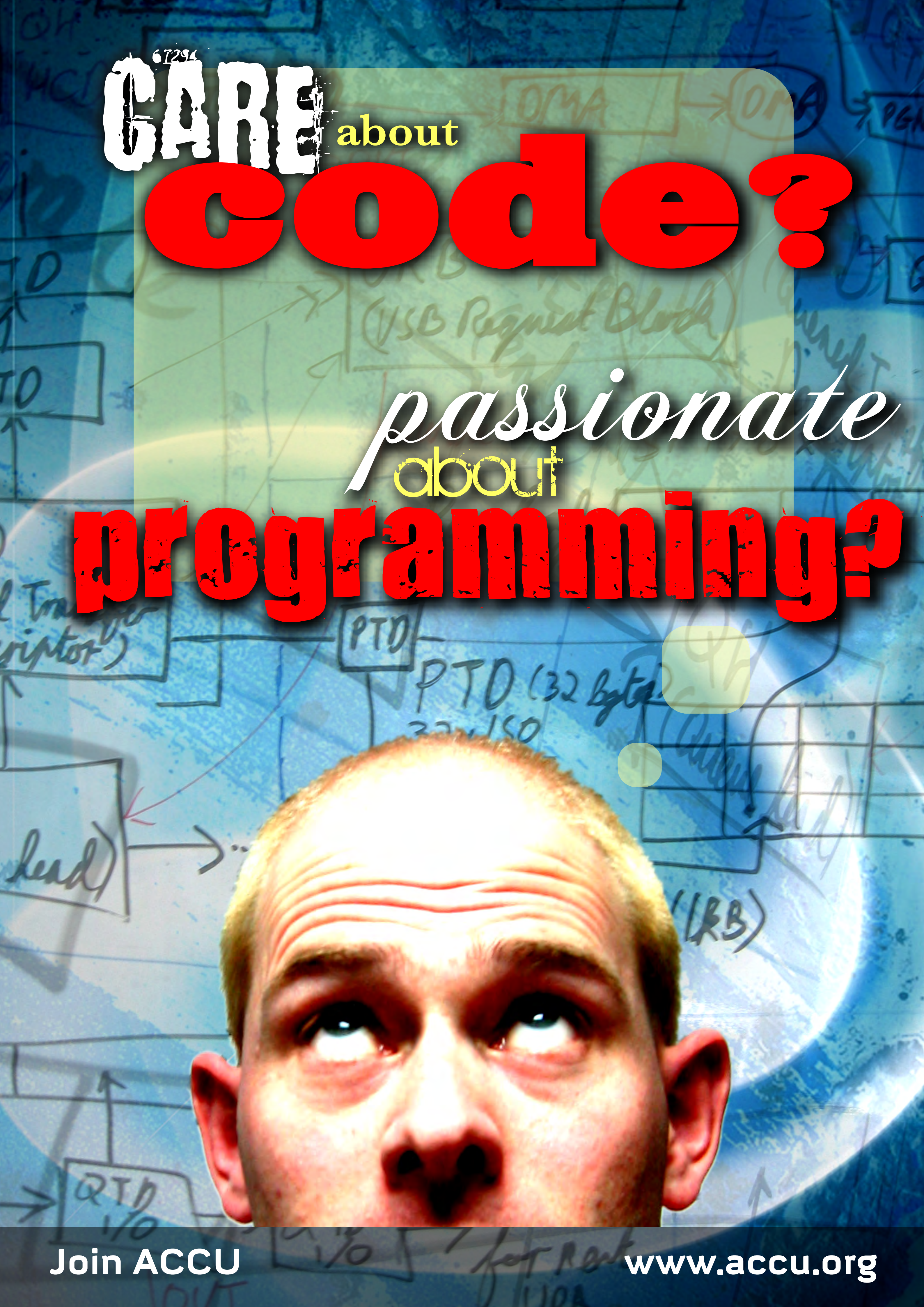
So would I want to go back to the way things were? No way! Maybe I'm just a Millennial who was born a decade too early but I'm relishing this ever decreasing feedback loop and I'm sure that as a result my programming output will only get better. ■

Reference

[Weinberg71] Gerry Weinberg (1971) The Psychology of Computer Programming, Dorset House Publishing Co Inc.,U.S.; Silver anniversary edition (29 April 1998) ISBN 978-0932633422

[Wikipedia] 'Jiminy Cricket', https://en.wikipedia.org/wiki/Jiminy_Cricket

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80's 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



CARE about
code?
passionate
about
programming?
Join ACCU
www.accu.org

GET MOORE
intel Software
intel
PARALLEL STUDIO XE
£634.99
TOOLS THAT EXTEND MOORE'S LAW
CREATE FASTER CODE—FASTER
Take your results to the next level with screaming-fast code.

QBS Software Ltd is an award-winning software reseller and Intel Elite Partner

To find out more about Intel products please contact us:

020 8733 7101 | enquiries@qbssoftware.com
www.qbssoftware.com/parallelstudio

qbs SOFTWARE