overload 153

OCTOBER 2019 £4.50

C++ Pipes

Expressive code makes life easier. We demonstrate fluent pipelines for data collections in C++

OOP Is not Essential

Object Oriented Programming still garners mixed reactions

I Come Here Not To Bury Delphi But To Praise It

What helps a programming language gain traction?

Scenarios Using Custom DSLs

A simple code-based alternative to natural language BDD

A magazine of ACCU

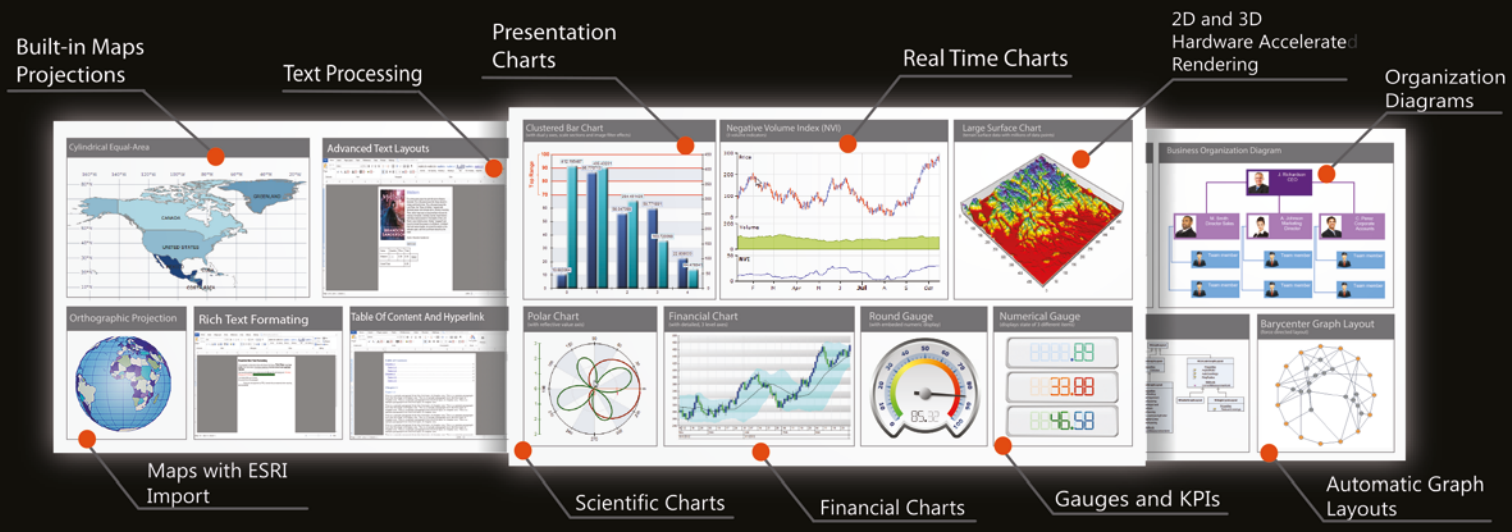
ISSN: 1354-3172

NEVRON

Visual Studio Partner

Nevron Data Visualisation and UI Controls

The leading data visualisation and UI components for desktop and server applications since 1998.



solutions available for

Microsoft ASP.net | WPF | WinForms | SharePoint | Microsoft SQl Server | Xamarin

View all of our Nevron products at www.qbssoftware.com/nevron

Key partners include:

Agisoft · ASPOSE File Format APIs · ATLASSIAN · axure · BrowserStack · DevExpress · embarcadero · Hex-Rays · intel Software · JetBrains Platinum Reseller · Microsoft Silver Partner · qbs Publishing · Sketch · SMARTBEAR · SPARX SYSTEMS · Telerik Develop experiences · think-cell · Visual Paradigm

Plus many more on **www.qbssoftware.com/developer**

For your latest software needs, contact our team on:

020 8733 7100 sales@qbs.co.uk

qbs SOFTWARE

OVERLOAD 153

October 2019

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Jon Wakely
accu@kayari.org

Anthony Williams
anthony.ajw@gmail.com

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 154 should be submitted by 1st November 2019 and those for Overload 155 by 1st January 2020.

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

Predictions and Predilections

Forecasting the future is difficult. Frances Buontempo has a foreboding sense that a lack of impartially makes things even harder.

Predicting whether or not *Overload* will have an editorial while I am the editor is easy. I attended the *Agile 2019* conference this year, co-chairing the Dev Practices and Craft track with Seb Rose [Agile]. This long trip, just for a few days, has increased my carbon footprint and decreased the time I had to think of an editorial. This means, yet again I haven't written one. Your prediction was correct: *Overload* is editorial-free yet again.

How do you make a prediction? Listing all the possible outcomes and assigning a probability to each gives a sense of the likelihood of a specific outcome. This approach has two problems: listing the outcomes and getting accurate probabilities. There are several formal approaches, such as Bayes, for working out these probabilities. In one sense the probability represents the uncertainty of a given event. Uncertainty can come in two flavours: epistemological, or due to lack of knowledge, and aleatory, or due to inherent randomness (see 'Bayesian statistics' [Spiegelhalter09] for more background]. If you sure you are uncertain, are you sure how uncertain you are? In other words, how do you decide how accurate these probabilities are? Some experiments can help, but you may not be able to try something a statistically valid number of times first. Furthermore, how do you predict how likely something is that has never happened before? Tough question to answer, but people try to do this. If you have no empirical data (what other types of data are there?), how can you guess how often something might happen? One approach is a forecasting model [Gelman98]. This requires a model, obviously. This might fit known cases, but I still find it difficult to accept forecasts around previously unseen events. I do understand the maths, but it all seems inherently odd.

Physics models, though often inspired by data, often take the form of a closed-form formula, rather like a function, taking inputs and returning a single output, rather than several outputs with confidence intervals. Sometimes models have been calibrated to data at some point, to find parameters, such as acceleration due to gravity. Some models are based on a combination of other known models. If all the forces acting on a body can be calculated, the total force can be deduced. In other domains, the idea of a straight summation breaks down. For example, if two chemicals have a known toxicity, the level of harm cannot be worked out by adding the two numbers. They may interact, and be less toxic overall, or even worse in conjunction. Going back to physics, though the motion of two bodies, such as the sun and a planet can be calculated, the three-body problem [Wikipedia] says that there is no closed-form solution for finding the motion of three, or more, objects, given their starting velocity and positions. Numerical methods are required instead. Or as I put it, "left a bit, right as bit" until you get something close enough to an answer. For some definition of close.

Not all prediction systems are based on statistics or models. Decision trees fall under the umbrella term machine learning. They give classifications of new data based on summaries of training data, in the form of flow charts or list of rules. Something like

> If Utility module updated on a Monday then build broken all week.

In order to find the tree or rules, some decision trees use entropy, which is formally an Information Theory idea. At a high level, it measures the chaos present in a system. If you toss a fair coin, you would expect it to be heads about half the times, and tails the remaining times. This is higher entropy, or more chaos, and making it hard to predict what the next toss will be. In contrast, if an unfair coin always comes down heads, every single time, it is much easier to predict accurately what will happen. Less chaos means you can compress this down very easily. In the first case, of a fair coin, writing a function to predict what happens next is harder and needs more lines of code. In the second case, the function need only return "Heads" each time. In a sense, this is still based on counting possible outcomes, but is taking a different perspective. I recently wrote a short blog post about decision trees [Buontempo19a]. At the expense of repeating myself, armed with rows of data, each with features and a category – either yes/no, heads/tails or one of many classes – you can build a classifier, which will tell you which category new data falls into. There are various ways to decide how to split up the data, including entropy. Regardless of the method, each algorithm follows the same overall process. Start with a root node then

1. If all the data at a node is in the same category (or almost all in the same category) form a leaf node.
2. For a non-leaf node, pick a feature, according to your chosen method.
3. Split the data at this node, some to the left branch, and some to the other branch (or branches) depending on the value of the chosen feature.
4. Continue until each node is a leaf node.

This is a bit like a sorting algorithm: in quick sort, you choose a pivot value and split the data down one branch or the other, until you have single points at nodes. Here we don't choose a pivot value but features. For example, is the coin heads or tails? The way to pick a feature can be based on statistics, information theory or even at random. At each step, you want to know if all the items in one category tend to have the same value or range of values of a feature. Once you are done you have a tree (or flow chart) you can apply to new data. Each way to split has various pros and cons. You can even build several trees. A random forest will build lots of trees and they vote on the class of new, unseen data. You could build your own voting system, using a variety of tree induction techniques. This might avoid some specific problems, like over-fitting from some

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.

techniques. Decision trees can be used to spot what features problematic scenarios have in common. Maybe all your bug reports end up with a fix in the same module. That might not be immediately clear until you analyse the data. If you want to know what certain things have in common, a decision tree is worth a try. My *Genetic Algorithms and Machine Learning for Programmers* book [Buontempo19b] has a chapter on building a decision tree from scratch if you want some details, but there are plenty of frameworks out there that will automatically build one for you. You may not be able to predict the future with your tree, and your machine may learn nothing, as is often the case in Machine Learning, however, you may spot something of interest.

Every classifier, be that a decision tree or not, has a set of possible outcomes, classes or categories. In various disciplines, including machine learning, the possible outcomes are described as a "search space". These can be more generally useful than for classifiers. When we moved house, we kept our cat shut in for a bit, so he could get the hang of his new home first, before exploring the great outdoors. He explored by trying a corridor and then returning to his starting point. Then he went a bit further, but always went back to the starting point. Each iteration added a small new part of his search space. In this case, the constant returning to the base station wasn't a bias, it was sensible. In a fit of laziness, we bought a Roomba, robot vacuum cleaner. This uses a similar algorithm. It has a base station, which it tends to stick near initially, gradually adding paths round the room, often making its way round the outside edges, just like our cat. I wonder if we could use the cat to sweep the floor. Nah, bad idea. I think I can see a few potential problems with that. This exploration of possibilities, though not predicting the future, includes an element of premonition. "Going forwards here means hitting a wall." Does learning mean you think you know what outcome is likely, given an initial set of conditions and a specific choice? Maybe. What do you think learning means? That's quite a big thing to think about.

Now, a spatial search brings various extra ways to make predictions. Sometimes you can guess where something might be, like mugs in a kitchen. There are often within reaching distance of a kettle. A combination of logic and expectations can be used to make an initial guess. I wonder if we tend to apply the same heuristics when dealing with code. Which header is `std::vector` in? What about `std::map`? Easy. What about `std::less`? That's another matter.

Predictions are often driven by some kind of bias. Why would you look in the fridge for dishwasher tablets? Because the light comes on when you open the door, so it's easier to see. Sense does not always prevail. More sensibly, if you plan a journey and hate public transport, you are more likely to consider driving, cycling, getting a taxi or walking over some other modes of transport. I wonder if all the recent AI research into self-driving cars is somewhat biased. I have maintained for a long time that Star Trek's transport technology would be far better. I presume this might be less polluting. It certainly wouldn't need upkeep of roads. And I don't recall any episodes where transporter accidents involve innocent pedestrians or cyclists. Not to say transporter accidents are unheard of. I just maintain someone, somewhere, has a predilection for cars and that is driving, pun intended, the research into modes of transport in the wrong direction.

I love AI, and find its twists and turn through history fascinating. Trying to predict where it will move in the futures is very difficult. As with much research, it is partially driven by those who fund the work, which is turn might be biased towards return on investment rather than usefulness or some kind of inherent value, whatever that means. Other technological innovations are, frankly, more disquieting. Recent stories of facial recognition in use at King's Cross station in London have causes questions and possible GDPR related issues. It seems the company responsible, Argent, claims the system will 'ensure public safety' [BBC]. I have questions. If it recognises faces, does it have a database of faces of people who should be arrested on sight? I would imagine if you tracked individuals' paths through the station, you may spot bottlenecks and bad signage and be able to improve the situation; however, this would not require saving people faces. Indeed, this immediately made me think of a variety of sci-fi stories, including *Face Off* and *Minority Report*. This possibly tells you more about my background and point of view than the event itself. Here's a conjecture:

> Your predictions tell me more about your history and bias than they do about the future.

Many physics models are based on odd theological or Weltanschauung (world-view) assumptions. How many colours are there in a rainbow? An English rainbow has seven, apparently because Newton regarded seven as a mystically significant number. Other cultures have different counts of the colours. Pythagoras refused to believe in irrational numbers, resisting them. He also felt circles were significant, so planets had to be spheres, and orbits had to be spherical. The starting assumptions colour the final models. Climate-change deniers are also working on a set of assumptions and biases. How to notice a bias, or predilection, underpinning a model or prediction is a hard question. Sometimes the predictions are close enough or the model seems to work, which might allow incorrect, or at least suspect starting points to slip through. Question your own assumptions when you next make a prediction. What point of view are you operating from? What does the world look like through someone else's eyes?

One final question. Why are you trying to make a decision anyway? Frequently, predictions are made in order to aid decision making. For example, guessing if it will rain will help me decide if I will need an umbrella. Figuring out what could possibly go wrong can help prepare for the worst. However, an impending sense of doom can lead to self-fulfilling prophecies. Can you predict the future without influencing it? Thinking through what might happen can be useful, though. Being accurate isn't the most important thing. Don't forget:

> A completely predictable future is already the past.
> ~ Alan Watts

What does matter is being aware of possible outcomes, probable contributing factors, and recognizing your assumptions. Bias in, bias out. A sense of wonder and enquiry in, endless possibilities and hope out.

References

[Agile] Agile 19 conference: https://www.agilealliance.org/agile2019/

[BBC] 'Data regulator probes King's Cross facial recognition tech', posted 15 August 2019 at https://www.bbc.co.uk/news/technology-49357759

[Buontempo19a] Frances Buontempo (2019) 'Decision trees for feature selection', posted on http://buontempoconsulting.blogspot.com/2019/07/decision-trees-for-feature-selection.html

[Buontempo19b] Frances Buontempo (2019) 'Genetic Algorithms and Machine Learning for Programmers', https://pragprog.com/book/fbmach/genetic-algorithms-and-machine-learning-for-programmers

[Gelman98] Andrew Gelman, Gary King, and John Boscardin (1998) 'Estimating the Probability of Events that Have Never Occurred: When Is Your Vote Decisive?' *Journal of the American Statistical Association*, 93 pp1–9. Accessed via https://gking.harvard.edu/files/gking/files/estimatprob.pdf

[Spiegelhalter09] David Spiegelhalter and Kenneth Rice (2009) 'Bayesian Statistics', published on Scholarpedia, available from: http://www.scholarpedia.org/article/Bayesian_statistics

[Wikipedia] Three-body problem: https://en.wikipedia.org/wiki/Three-body_problem

Scenarios Using Custom DSLs

Natural-language BDD can be hard to maintain. Liz Keogh demonstrates a simple code-based alternative.

One of my clients recently asked me how often I use Cucumber or JBehave in my own projects. Hardly ever, is the answer, so I want to show you what I do instead.

The English-language Gherkin syntax is hard to refactor. The tools form another layer of abstraction and maintenance on top of your usual code. There's a learning curve that comes with them that can be a bit tricky. The only reason to use the tools is because you want to collaborate with non-technical stakeholders. If nobody outside your team is reading your scenarios after automation, then you don't need them.

There may still be other reasons you *want* the tools. They'll be more readable than the code I'm about to show you. Dynamic languages are harder to refactor anyway; I work primarily with static typing. Maybe you want to take advantage of hooks for your build pipeline. Maybe you already know and feel comfortable with the tools. Maybe you just really want to learn the technique. That's OK. But you don't need them.

So here's a simple alternative.

Have some conversations, and write down the examples

I like it when the developers do this, and get feedback on their understanding. Writing it in semi-formal Gherkin syntax is pretty useful for helping spot missing contexts and outcomes. All the usual goodness of Three Amigos conversations still applies.

Find a capability, and the thing that implements it

Your application or system probably has a number of things that it enables people or other systems to do. We're going to be using a noun that matches those things as a way of starting our DSL. Here are some examples:

- Buying things → the basket
- Making trades → a trade
- Commenting on an article → a comment / the comments
- Doing banking → the account

You may find the language is a bit stilted here (I did say the English was clearer!) but that's a trade-off for the ease of getting started with this. You might find other things which make more sense to you; it's sometimes possible to use verbs for instance.

- Searching for a car → I search

Liz Keogh is a Lean and Agile consultant based in London. She is a well-known blogger and international speaker, a core member of the BDD community and a passionate advocate of the Cynefin framework and its ability to change mindsets. She has a strong technical background with 20 years' experience in delivering value and coaching others to deliver, from small start-ups to global enterprises. Most of her work now focuses on Lean, Agile and organizational transformations, and the use of transparency, positive language, well-formed outcomes and safe-to-fail experiments in making change innovative, easy and fun. Contact her on tiwtter at @lunivore

You'll get the idea in a moment. Each of these is going to be the stem of a bit of code.

Start with comments in the code

Sometimes I like to just start with my scenario written in comments in the code. For each step, think about whether the step has already happened, is the thing that triggers some interesting behaviour, or is the outcome of that behaviour. Add Given, When or Then as appropriate:

```
// Given an article on Climate Change
// When I post a comment "This is a really
// conservative forecast."
// Then it should appear beneath the article.
```

Add the Given, When or Then to your stem, and...

- Given the basket…
- When the trade…
- When a comment…
- When a search…
- Then the account…

...construct your steps!

Now we're in code.

```
GivenAnArticle().on("Climate Change")

GivenTheBasket().contains("Pack of Grey Towels")

WhenTheTrade().isCreated()
  .withCounterparty("Evil Corp")
  .forPrice(150.35, "USD")
  .....
  .andSubmitted()

WhenISearch().For("Blue Ford Fiesta")

ThenTheAccount().shouldHaveBalance(15.00, "GBP")
```

You can see that trading one is using a builder pattern; each step returns the trade being constructed for further changes, until it's submitted. I sometimes like to use boring, valid defaults in my builder so that these steps only call out the really interesting bits.

I normally suggest that a 'When' should be in active voice; that is, it should show who did it. If that's important, add the actor.

```
WhenTheTrade().isCreated()
  .....
  .andSubmittedBy("Andy Admin")
```

or

```
WhenTheTrade().isCreated()
  .by("Andy Admin")
  .....
  .andSubmitted()
```

```
public class Scenario
{
  private readonly SudoqueSteps _sudoqueSteps;
  private readonly CellSteps _cellSteps;
  private readonly HelpSteps _helpSteps;

  private World _world;

  protected Scenario()
  {
    _world = new World();
    _sudoqueSteps = new SudoqueSteps(_world);
    _cellSteps = new CellSteps(_world);
    _helpSteps = new HelpSteps(_world);
  }

  protected CellSteps WhenISelectACell{ get
  { return _cellSteps; }}

  protected CellSteps ThenTheCell{ get
  { return _cellSteps; }}

  protected SudoqueSteps GivenSudoque{ get
  { return _sudoqueSteps; }}

  //...

  protected HelpSteps WhenIAskForHelp { get
  { return _helpSteps; } }

  protected HelpSteps ThenTheHintText { get
  { return _helpSteps; } }
}
```

Listing 1

Active voice would normally look more like:

```
When("Andy Admin").createsATrade()
  ....
  .andSubmitsIt()
```

But now our 'When' is ambiguous; we can't tell which kind of capability we're about to use, so it makes it really, really hard to maintain. It's OK to use passive voice for DSLs.

As I construct these, I delete the comments.

Sometimes I like to just put all the detailed automation in which makes the steps run, then remove the duplication by refactoring into these steps. (Sometimes it's enough just to just leave it with detailed automation, too, but at least leave the comments in!)

Pass the steps through to Page Objects; use the World for state

You'll probably find you need to share state between the different steps. I normally create a 'World' object, accessible from the whole scenario.

Each of the stems you created will correspond to one or more page objects. I like to keep those separate, so my steps in the DSL don't do anything more than just call through to that object and return it.

Listing 1 is an example of my scenario object for a Sudoku solver.

```
[TestFixture]
public class PlayerCanSetUpAPuzzle : Scenario
{
  [Test]
  public void APlayerCanSetUpAPuzzle()
  {
    GivenSudoque.IsRunning();
    WhenISelectACell.At(3, 4).AndToggle(1);
    ThenSudoque.ShouldLookLike(
      "... ... ..." + NL +
      "... ... ..." + NL +
      "... ... ..." + NL +
      "           " + NL +
      "... ... ..." + NL +
      ".1. ... ..." + NL +
      "... ... ..." + NL +
      "           " + NL +
      "... ... ..." + NL +
      "... ... ..." + NL +
      "... ... ..." + NL);
  }
}
```

Listing 2

It does get quite long, but it's pretty easy to maintain because it doesn't do anything else; all the complexity is in those underlying steps.

And Listing 2 shows how I use it. Full scenarios are available at https://github.com/lunivore/sudoque/tree/master/Sudoque.Scenarios.

This one was written in plain old NUnit with C#. I've done this with JUnit and Java, and with JUnit and Kotlin. The examples here are only from toy projects, but I've used this technique on several real ones.

There are lots of tools out there which help you to construct these kind of DSLs; but I've found they also come with their own learning curve, constraints, maintainability issues etc.. This is a pretty easy thing to do; I don't think it needs anything more complicated than I've put here.

It's also very easy to refactor overly-detailed, imperative scenarios, of the kind created by lots of teams who didn't know about the conversations, into this form.

It's easy to move to the BDD tools if you need them

With your Page Objects already in place, it's pretty quick to get something like Cucumber up and running and make the step definitions call through to the page objects exactly as you were before, with just a little bit of refactoring of method names.

It's a lot harder to move from Cucumber and regex to a DSL.

Chris Matts once had some great wisdom. "If you don't know which technology to choose, pick the one that's easy to change. If it's wrong, you can change it."

This is the one that's easy to change, so I tend to start with this. And sometimes it doesn't need to change. ■

This article was first published on Liz Keogh's blog: https://lizkeogh.com/2019/08/27/scenarios-using-custom-dsls/.

OOP Is not Essential

People tend to love or hate Object Oriented Programming. Lucian Teodorescu considers a recent OOP claim.

A recent article from Ilya Suzdalnitski [Suzdalnitski19] complained of the 'disaster' that OOP has become, compared to its promise. The article received quite a bit of attention, both on Medium and on Twitter. As the article was rather more ideologic than argumentative, the reactions ranged from very positive to very negative. One particular reaction that caught my attention was from Grady Booch [Booch19].

Besides being from Grady Booch (a prominent figure in the OOP world), the thing that most caught my eye was the "multitudes of real world systems for which object-orientation was essential" statement. As yet, I have not seen any compelling examples making OOP necessary. Moreover, I believe that such examples do not exist, and the present article is an attempt to show why.

It is far from my intent to pick on some wording that Booch used (in some less-formal context). What I would like to argue against is the common belief that Object-Oriented Programming is the 'true' way of writing any software system.

But, by all means, if somebody has such a list of examples of real-world projects in which OOP was/is essential, please share it with me. And, for that matter, of any programming paradigm. Any argumentative explanation of the form 'software X essentially needs programming paradigm Y' would most likely advance our studies on software engineering.

Lucian Radu Teodorescu has a PhD in programming languages and is a Software Architect at Garmin. As hobbies, he is working on his own programming language and he is improving his Chuck Norris debugging skills: staring at the code until all the bugs flee in horror. You can contact him at lucteo@lucteo.ro

The meaning of 'essential'

In the context of the tweet, we can distinguish three possible meanings for the 'essential' word:

- as in Brooks' division between essential and accidental [Brooks95] – i.e., there are software systems in which their essential complexity somehow mandates OOP (see below)
- with the meaning of 'necessary'– i.e., the software system cannot be built without it
- with the meaning of 'it's much easier with' – i.e., building the software system is much easier with OOP; it can be built without OOP but with much higher costs

Let's analyze how OOP can be (or not be) essential in a software system from all three perspectives.

Brooks' essential

Brooks makes the following division:

> […] to see what rate of progress we can expect in software technology, let us examine its difficulties. Following Aristotle, I divide them into essence – the difficulties inherent in the nature of the software – and accidents – those difficulties that today attend its production but that are not inherent.

He then immediately goes to say:

> The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract, in that the conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed.

And then he describes what he believes is the irreducible essence of modern software systems: complexity, conformity (to existing interfaces), changeability and invisibility.

Nothing in what Brooks calls essential is fundamentally attacked by OOP. Furthermore, Brooks has a small section on Object-oriented programming, in which he states that OOP attacks accidental difficulties:

> Nevertheless, such advances can do no more than to remove all the accidental difficulties from the expression of the design. The complexity of the design itself is essential; and such attacks make no change whatever in that.

In the '"No Silver Bullet" Refired' chapter [Brooks95], Brooks remarks that, after 9 years since the original claims, OOP has grown slower than people would believe.

Ok, so clearly OOP is not essential for any software system in the way Brooks describes 'essential'.

'Essential' as 'necessary'

Let us assume Booch intended to say "multitudes of real world systems for which object-orientation was necessary", with the meaning that the software could not be technically written without OOP. Similar to saying that the complexity of a sorting algorithm is essentially $O(n \log n)$ – that is,

The fact that people are biased towards using OOP doesn't make OOP essentially simpler than other programming paradigms

in the general case, the order of magnitude for the number of comparisons cannot be less than $n\log n$.

But that cannot be the case. Any software system that can be built using one programming language/paradigm can be built using another language or paradigm. After all, all the programming languages and, by extension, all programming paradigms are Turing-complete (any programming paradigm can be used to implement Turing-computable functions).

'Essential' as a form of simplicity

In the last meaning that we explore, we assume that Booch wanted to say "multitudes of real world systems for which object-orientation makes the problem much easier to solve". This is starting to sound a bit more plausible.

A statement like "Project X can be solved by team Y with OOP simpler than it can be solved in any other programming paradigm" is a fair statement. I think most of the readers will agree with it.

But, we must argue that we cannot generalize it for all the teams. OOP is not necessarily the simplest way to write (reasonably complex) software. For example, consider people like Joe Armstrong (creator of Erlang, who sadly died this year) [Seibel09], [Armstrong], Linus Torvalds (who expresses so colorfully his dislike of C++/OOP) [Torvalds04,07]], Simon Peyton Jones (designer for Haskell) or Rob Pike (designer for Go). Would they consider that OOP is the easiest method to write software? Definitely not.

Different people and different teams will have different proficiency levels with different technologies/paradigms. Out of all the factors that affect the proficiency of an individual/team, probably education is the most important one. If the industry highly esteems OOP programmers, if most of the formal education encourages people to believe that OOP is the most important programming paradigm, and if most of the software literature teaches OOP, then, of course, people will start to be proficient in OOP, and become biased towards OOP. (It is hard to generalize, but my personal opinion is that OOP is still highly promoted, probably more than it merits.)

The fact that people are biased towards using OOP doesn't make OOP essentially simpler than other programming paradigms. It's probably just confirmation bias. People with hammers see nails all around, which strengthens their belief that the hammer is the best tool.

With all these said, we can conclude that even within this interpretation, OOP is not essential in building software systems.

OOP features

Let us now analyze the problem from a different perspective; let us try to answer the following question: Is there some OOP feature that is not present in other programming paradigms and that would help the programmers better tame the complexity?

We will analyze the major features of OOP to answer this question. When I say OOP, I'm mainly thinking of languages like C++, Java, and C#. I will often contrast them with non-OOP languages like C and with functional languages (Haskell, ML)

Objects and classes on top of imperative programming

OOP is an imperative paradigm. Nothing new here. It has classes and objects, but those aren't necessarily something new.

Classes, in the absence of encapsulation, are just data structures. Similar to C structs; similar to product types in functional paradigms. Objects are instances of these classes – in other words, values. There is nothing new that OOP adds here to help in dealing with complexity.

Please note that OOP has a convention that classes, i.e., data structures, should correspond to things in the real world. I find this a bit disturbing, but that is not the issue here. There is nothing that prevents other paradigms from adopting similar conventions.

Things like class variables are just syntactic sugar. There is nothing here that essentially helps in fighting complexity.

Encapsulation

Encapsulation is the concept that binds together the data and the functions that manipulate that data. As opposed to traditional imperative programming, OOP puts functions inside classes, and calls them methods. But, a method is nothing more than a function that takes the object as a (hidden) parameter. Everything is syntactic sugar.

Nothing prevents a C programmer from placing all the functions that operate on the data near the struct definition. Ignoring the access rights of methods and attributes, this convention produces similar results. With any programming language that supports some sort of package constructs, one can easily emulate encapsulation. Signatures and structures in ML (functional language) [Harper00] behave very similar to encapsulation in OOP.

Again, nothing that cannot be done with simple conventions; at best, improvements that OOP adds here would fall into fighting accidental difficulties.

Now, let me be clear about one point. In general, encapsulation can be seen from two different perspectives:

- a syntactic perspective, on how OOP languages recommend placing data and functions together
- a modeling perspective, a way of thinking about programs, that tends to put data and the operations on the data together

The argument here was at the syntactic level. OOP languages add syntactic sugar to easily allow programmers to group data and functions.

The most important part of the encapsulation comes with modeling perspective. That is a form of decomposition that can actually help fight complexity (see below in the 'What is truly essential?' section). But again, nothing can prevent a C or an ML programmer from using this way of thinking about problems. So, even though this modeling technique is typically associated with OOP, other non-OOP languages can use it.

Information hiding

Preventing the programmer from accessing some variables/functions can hardly be called an essential improvement for software engineering. If one

there is no single OOP feature that would have significant importance in fighting essential complexity, or in making programmer's life much more easier

needs help in hiding that information, one can always rely on packaging systems, on conventions (like the leading underscore in Python) or even code documentation.

Polymorphism

People often claim that OOP is needed to have polymorphic behavior. Nothing can be more false than that.

First, let us acknowledge the existence of multiple types of polymorphism: subtype polymorphism (or inheritance based – the one advertised by OOP), ad hoc polymorphism (i.e., overloading) and parametric polymorphism (as used by functional programming languages, but also for implementing generics/templates in some highly acclaimed OOP languages). And, there is also duck-typing, a form of polymorphism without static types.

There are no technical reasons to believe that subtype polymorphism is superior to parametric polymorphism. On the contrary, I believe the opposite; but I'll leave that discussion for another time.

Also, the reader should consider that basic polymorphism can be constructed in C with manual vtables. OOP languages just add syntactic sugar on top of this.

As polymorphism is not unique to OOP, we also conclude that, with respect to polymorphism, OOP cannot be essential in building software.

Inheritance

We have already discussed how subtype polymorphism present in OOP is not essential to building software. Without the polymorphism aspect, inheritance is drained out of substance. There are voices that claim that inheritance is abused, and there are a lot of cases in which it can be replaced by simple composition. For example, see item 34 (Prefer composition to inheritance) from *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices* [Sutter04] and the *Inheritance Is The Base Class of Evil* presentation [Parent13].

Without polymorphism, inheritance is just syntactic sugar (one that can cause harm if abused).

Dynamic dispatch

Although object-oriented languages provide an easy method of implementing dynamic dispatch (e.g., virtual functions in C++), other languages have different strategies. Languages like C provide function pointers to handle this, while functional languages provide closures to implement dynamic behavior. Essentially, a closure or a function pointer is an interface with a single method, and any object-oriented interface can be decomposed into smaller, one-function interfaces.

Yes, interfaces with multiple (virtual) functions can be slightly more efficient in some contexts, but there is nothing game-changing in having multiple functions per interface. In the worst case, the user can group multiple one-function interfaces into one single data structure.

Again, OOP doesn't provide a feature that is unique and cannot be matched with a bit more syntactic verbosity; and we already established that is an accidental difficulty, not an essential one.

In summary

There is no single OOP feature that would have significant importance in fighting essential complexity, or in making programmer's life much more easier. They can be all summed up in the category of syntactic sugar.

Syntactical and modeling – an analogy

Most of the discussion about OOP features revolved around syntactical aspects of OOP. The reader should be guessing by now that, following Brooks, I have a strong position for dismissing syntactic features as solving accidental difficulties, and not being essential to the development process. Yes, it can make you write 10% faster code, but it is not essential.

There is a different story with the modeling perspective of OOP. We'll tackle this in the next section. But before that, I want to draw an analogy.

Let's compare OOP with the traditional motor car (with internal combustion). While there are a lot of cars out there, the cars are not essential to locomotion. We can travel by plane, we can travel by boat, we can travel by train, we can travel on a horse and even by foot.

Similar to the syntactical features of OOP languages, we can think of the shape of the car. It is true that the cart alone doesn't make the car; it just adds marginal improvement to locomotion (faster speeds, better grip, etc.)

The modeling aspect of OOP is analogous to the internal combustion engine of the car. The engine is what makes the car a car. But what it *is* important here to notice is that there are alternatives to internal combustion engines. We have fully-electric engines, we have hybrid-engines, we have engines based on wind or even on solar power; and let's not forget horses and locomotion by foot.

The main point is that neither the cart nor the internal combustion engine is essential for locomotion. And internal combustion engines, even though they are most commonly seen on cars, can be present on other locomotion machines.

In the previous sections we went over major OOP features. We concluded that most of them are syntactic features. The modeling aspects that are commonly found on OOP languages (encapsulation, polymorphism) can be present in non-OOP languages. But are these OOP modeling techniques essential?

What is truly essential?

Decomposition. The breaking down of a complex software system into multiple parts that are easier to understand, to reason about and to maintain. Only by decomposition can one hope to tame the complexity.

But beware, decomposition can fall into the same bias as discussed above. We can say that a certain decomposition would make the software system easier to understand for team X, but we cannot say that it will do for any team/individual.

In OOP, people usually follow the so-called object decomposition: we try to break down the system around 'things' (as opposed to operations or functions), which will become objects/classes. As these objects/classes will hold state, this type of decomposition typically is a decomposition of state: the state will be scattered (and shared) around all over the software. This is typically a bottom-up approach. See also Booch method [Booch94]

By contrast, functional decomposition as found in functional languages considers functions as basic building blocks. It is more focused on decomposing data flows. The state is typically immutable and isolated (i.e., the inputs of a function are always distinct from the outputs of the same function). The pipes and filters pattern typically employs a functional decomposition. This type of decomposition mostly resembles top-down decomposition.

But, just because these two dominate OOP and functional programming, it doesn't mean that there aren't other types of decomposition. Here are some decompositions that can make a lot of sense, but not get that much attention: decomposition based on security levels, based on the distance from the user (think of a web, layered architecture), based on the expertise of different teams/individuals (Conway's law [Conway68]), etc.

In practice, in one software project, typically more than one decomposition appears. If one decomposition appeals to a group of people, it may not appeal another group of people, at least not at first sight. For example, I believe that a decomposition based on security levels is not something that most of the readers will think of first; on the other hand, I believe there will be other readers who apply it very frequently.

So, to come back to the previous analogy, there aren't only internal combustion engines. Fully-electric engines are starting to show a lot of potential (can this be similar to functional programming?). Plus, there are hybrid-engines which don't seem too bad (I find this analogous to C with encapsulation based on conventions). Let's not forget about non conventional engines, like wind-powered ones (to be associated with less prominent programming paradigms).

Just like engines, the different types of decompositions have pros and cons. Like there is no 'essential' engine, there isn't any programming paradigm that is essential to solving a software problem.

Conclusions

We argue here that OOP is not essential for software systems. It can be easier for certain teams/individuals, but we cannot generalize. The word 'essential' cannot be used in this context with the meaning that Brooks attributes to the word, and it cannot mean 'necessary'. In limited contexts, it can mean simpler; but this simpler is directly dependent on the people for which it is simpler – there is no such thing as simpler for everyone.

To make sure we haven't missed anything, we also looked at the problem from a different perspective: trying to see if there are some features of OOP that can promise simpler software. But almost all the important OOP features are merely syntactic sugar; all OOP programs can be translated into C with minimal effort.

The most important tool for solving software problems is decomposition. But this is not particularly tied to a programming paradigm. As an industry, we should probably be focusing more on different ways of decomposing a complex software system rather than trying to religiously apply one paradigm or another. There is nothing fundamental that would prevent a programmer from applying good decomposition principles in C as opposed to an OOP language.

But probably there are still readers that believe that C doesn't allow high-level abstractions. I would urge those readers to carefully analyze the reasons for that belief. I am highly convinced that similar to the content exposed in this article, the main reasons are:

- the language features that enable those high-level abstractions are just syntactic sugar – accidental difficulties
- the reluctance of creating high-level abstractions in a language like C comes from internal biases

To overcome the biases I recommend the readers to get exposed to multiple decomposition methods and multiple programming paradigms. With that in mind, I would also recommend the readers to go over Ilya's post [Suzdalnitski19]; it may be ideologic, it might not have all the proper arguments, but it offers a non-traditional perspective on software construction.

OOP may be helping a lot of people to write good software. But claiming essentialness of OOP is a bit too strong, in my opinion. ■

References

[Armstrong] Joe Armstrong, 'Why OO sucks', https://www.cs.otago.ac.nz/staffpriv/ok/Joe-Hates-OO.htm

[Booch19] Grady Booch (2019), Twitter reply to Ilya, https://twitter.com/Grady_Booch/status/1153176945951068161?s=17

[Booch94] Grady Booch (1994), *Object-oriented analysis and design with applications*, Addison-Wesley Professional

[Brooks95] Frederick P. Brooks Jr (1995), *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition (2nd Edition), Addison-Wesley Professional

[Conway68] Melvin Conway (1968), 'How Do Committees Invent?' http://www.melconway.com/Home/pdf/committees.pdf

[Harper00] Robert Harper (2000), 'Signatures and Structures' in *Programming in Standard ML*, https://www.cs.cmu.edu/~rwh/introsml/modules/sigstruct.htm

[Parent13] Sean Parent (2013), 'Inheritance Is The Base Class of Evil' at GoingNative 2013, available from: https://www.youtube.com/watch?v=bIhUE5uUFOA

[Seibel09] Peter Seibel (2009) 'Joe Armstrong' in *Coders at Work: Reflections on the Craft of Programming*, Apress

[Sutter04] Herb Sutter, Andrei Alexandrescu (2004), *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*, Addison-Wesley Professional

[Suzdalnitski19] Ilya Suzdalnitski (2019), 'Object-Oriented Programming – The Trillion Dollar Disaster', https://medium.com/better-programming/object-oriented-programming-the-trillion-dollar-disaster-92a4b666c7c7

[Torvalds04,07] Linus Torvalds (2004, 2007), Linus Torvalds on C++ (correspondence), http://harmful.cat-v.org/software/c++/linus

I Come Here Not to Bury Delphi, But to Praise It

What helps a programming language gain traction? Patrick Martin remembers why he used to use Delphi.

But first what is Delphi?

It is a riddle, wrapped in a mystery, inside an enigma.

And why write about it?

It's not a controversial statement that Delphi is not what it once was in its heyday [OracleAtDelphi05]. Nevertheless, I think it's worth reviewing what might have formed part of the secret sauce that was part of its success back then. The current version now supports building 64-bit projects across the 3 main desktop platforms, to select one area of evolution of the product.

Furthermore the original aspects that were key for me are not only still there, but better than before.

Non goals and own goals

There are many things I will not be discussing in this article. For example, there is always Much To Discuss when it comes to choice of programming language – I am told – but I will be attempting to steer clear of controversy of that nature.

Comparing laundry lists of features between languages in some kind of checklist knockout tournament is certainly not the aim here.

Instead, I want to recall – or if you will, eulogise – a rich seam of features of the tool that for me made Delphi the game changer that it was then...

Back when I used it full time, these techniques were what made it so productive and what's more fun to work with, and I humbly submit that there are few tools that come close to touching it even now.

First, a quick review

Delphi is a commercial product for developing software [Embarcadero-1], [Wikipedia-1], with a proprietary IDE and version of Object Pascal [Wikipedia-2] that integrates tightly with the solution [Embarcadero-3]. There is even a free version you can download from [Embarcadero-2], and if you can puzzle your way past the registration djinns, you can have it installed and up and running in a few minutes.

Table 1 shows a heavily abridged table of releases with my comments for some key milestones.

For prior art: there is even an ACCU article [Fagg98] article, and if you want a much funnier, arguably less slightly less technical summary of the early days, try this on for size [Stob12].

Here are seven bullet points I've chosen to give a flavour of the system:

- **Fast**

  Compile times were always in the vanguard, currently there are quotes of many thousands of lines per second.

Year	Release	Supports development for	Notable Enhancement
1993	Delphi 1.0	Win16	From out of nowhere, handling a GPF*
1996	Borland Delphi 2	Win32	first win32 compiler
1998	Inprise Delphi 4	Win32	last version allowing 16-bit development
2003	Borland Delphi 8	Win32	.NET
2005	Borland Delphi 2005	Win32	
2011	Embarcadero Delphi XE2	Win32, Win64	first version producing 64-bit binaries
2012	Embarcadero Delphi XE3	Win32, Win64	last version with .NET support
2018	Embarcadero Delphi 10.3 Rio	Win32, Win64	current day

* What is a GPF? [Fahrni18]

Table 1

The time from a standing start of just source to a fully linked native executable that was ready to go was also very, very short. In the days of 'spinning rust' drives, this is a feature that really mattered – there is a nice little review here [Hague09].

- **Strongly typed** (mainly[1])

  `for` loops could only be Ordinal types. I got over the shock of not being able to increment a **`double`** type very quickly and never looked back.

  You could (and should) declare enums and sub-range types. It would then be a compilation *and runtime* error to assign incorrect values to these types, if you chose to enable the strict compilation mode, which you almost always should.

```
type
// everyone likes cards
  Suit = (Club, Diamond, Heart, Spade);
// small things that it's just embarrassing
// to get wrong
  SmallNumber = -128..127;
  SomeCaps = 'A'..'Z';
  Month = 0..11;
```

- **Run time type information at all times**

  One could always identify the type of an object at runtime and it was built into the language – with a little more effort one could browse all the types in the programs' type system. This will come in useful for building up complex objects, as we will see.

1. Although, there were some funky compiler features that allowed for late-bound function calls, mainly to support scripting OLE objects.

Patrick Martin Patrick's github repo was classified using a machine learning gadget as belonging to a 'noble corporate toiler'. He can't top that. Patrick can be contacted at patrickmmartin@gmail.com.

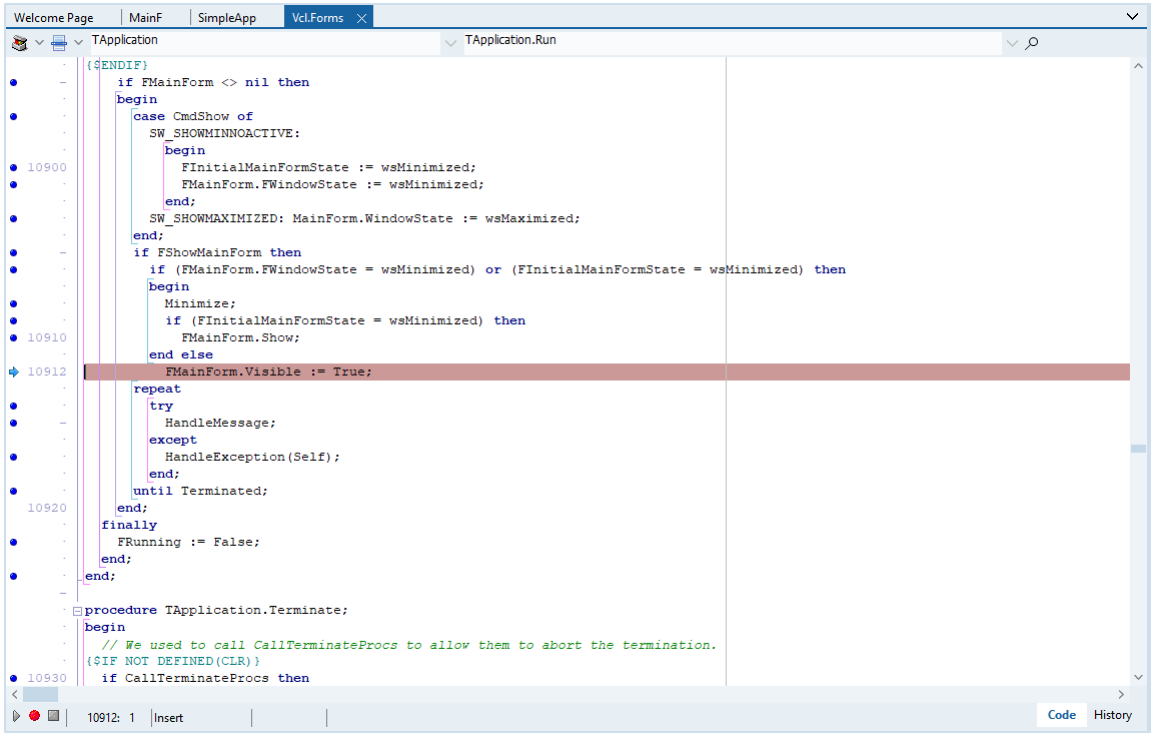


Figure 1

- **Straightforward dependency management**

  The language has files called units – basically modules, which supported interface and implementation sections for exported symbols and internal only code.

  Circular dependencies were a compile time error, it's worth taking a second to let that sink in.

  This required the developer to structure their program as a Directed Acyclic Graph, which strongly encouraged a way of organising one's code in such a way that one really only had to inspect the interface section of a new dependency unit, and then choose whether to make it a dependency in the implementation or not.

  Rinse and repeat for the rest of the program.

  In addition, the order of initialisation and finalisation of the units was straightforward and robust (even if spelled incorrectly – see later ☺).

- **Extensible RTL and Visual class libraries exploiting the strengths of the language**

  Object Pascal supports class properties (read/write, write-only, read-only) as a first class feature. Objects on the stack are simply not allowed – I suspect eliminating this capability freed Delphi from having to deal with a large class of issues related to dynamic runtime use of code. Coupled with the ability to use the RTTI, these work together to support configurability of classes from properties.

- **Source-based component model**

  It's worth bearing in mind that the time when Delphi was conceived was the era of the rise of the component-based software model [Wikipedia-4]. For example, people could pay (remember paying for software?) a nugatory amount for a component that would emulate Excel and embed it into their software.

  In the very first release of Delphi was a thorough guide to writing components, proselytising for the style of authoring components. This was really high quality work and – true story – we kept a copy of the Delphi 1 guide chapters that didn't survive to later releases around to consult, as it remained relevant.

- **You could go deep if you wanted**

  For those minded to use them, the features of a language for the 'hard core programmer' were also there:

  - full access to the FFI of binaries of other languages at link time
  - a range of selectable calling conventions (see [Wikipedia-3] for details of these)
  - capability to hand-craft dynamic loading of code modules
  - all the usual crazy casting stuff some programmers like to do (rarely needed in Delphi)
  - inline assembly.

Contention: Delphi was inherently very dynamic for its time

This is my central thesis.

In 1999, I could fire up the IDE, load the source of a visual form connected to, say a database and see and navigate records fetched from the database *live in the designer*. The development environment was quick and effective to work in, and I had access to the source for debugging and simply improving my mind by reading the code.

That feature just on its own, helped to teach me a lot about the engineering of a coherent architecture. And for those prepared to take the time to investigate, it had a cornucopia of treasures to uncover beyond the super user friendly surface.

Example: how the ability to read and *debug into* the source make a difference

Figure 1 is a simple UI app I created in a few clicks with no code. Setting one breakpoint and stepping in using one key combination I see where the application launches the main UI form and then enters the main Windows interactive message loop.

Remember the 90s were wild, man

This was the time of the rise of the Component based model – people could pay (remember paying for software?) a nugatory amount for a component that would emulate, say some portion of the Excel spreadsheet editor and embed it into their software [Wikipedia-4].

In Delphi I could study the built-in in components, or follow the tutorials and write my own if needs be, or figure out how to achieve my aims using the existing functionality.

Delphi's streaming system and form design

Now the real killer app for the app development was the fully synchronised visual designer

Let's have a look at some actual code to plug together some hypothetical framework objects. Note, this process relies upon the concepts of

- properties
- the Delphi closure type (reference to method call on object instance)
- and RTTI to allow the RTL to work all the magic of wiring up the properties
- there is also a hint of a framework which defines the ownership from the line `TfrmClock.Create(Application);`

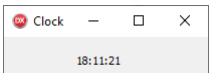
```
begin
  frmClock := TfrmClock.Create(Application);
  lblTime := TLabel.Create(frmClock)
  lblTime.Caption := '...'
  tmrTick := TTimer.Create(frmClock);
  tmrTick.onTimer = tmrTickTimer;
  frmClock.AddChild(tmrTick);
  // ...
end;
```

And let's have a look at some hypothetical DSL code to describe the moral equivalent of that code

```
object frmClock: TfrmClock
  Caption = 'Clock'
  object lblTime: TLabel
    Caption = '...'
  end
  object tmrTick: TTimer
    OnTimer = tmrTickTimer
  end
end
```

Full disclosure: Of course it's actual real DSL (edited slightly for space)! *The IDE would generate all of that for you.*

Now, with the mere addition of the following line to a class method called `tmrTickTimer` … we have a clock app!




```
lblTime.Caption := TimeToStr(Now);
```

So, that's assembling visual components visually sorted then.

Registry singletons done right (TM)

Listings 1–4 are an example illustrating how deterministic initialisation of modules would allow for very simple, yet very robust registration concepts, giving the following output:

```
Registry Adding: TSomeProcessor
Registry Adding: TAnotherProcessor
Program starting
Registration complete
Program exiting
Registry Removing: TAnotherProcessor
Registry Removing: TSomeProcessor
```

Note the initialisation follows the lexical ordering in the program unit *in this case* (but see later), and also that the de-init occurs perfectly in the inverse order.

Add this `uses` directive into `SomeProcessor`, adding a source level dependency to `AnotherProcessor` from the `SomeProcessor` *implementation* (Listing 5).

The output is:

```
Registry Adding: TAnotherProcessor
Registry Adding: TSomeProcessor
Program starting
Registration complete
Program exiting
Registry Removing: TSomeProcessor
Registry Removing: TAnotherProcessor
```

```
program registration;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  AnotherProcessor in
'depends\AnotherProcessor.pas',
  SomeProcessor in 'depends\SomeProcessor.pas',
  SomeRegistry in 'depends\SomeRegistry.pas';
begin
  try
    WriteLn('Program starting');
    WriteLn('Registration complete');
  except
    on E:Exception do
      Writeln(E.Classname, ': ', E.Message);
  end;
  Writeln('Program exiting');
end.
```

Listing 1

```
unit SomeRegistry;
interface
type
TSomeRegistry = class
public
  procedure RegisterClass(AClass: TClass);
  procedure DeregisterClass(AClass: TClass);
end;
function GetSomeRegistry: TSomeRegistry;
implementation
var
  mSomeRegistry : TSomeRegistry = nil;
// details omitted
initialization
  mSomeRegistry := TSomeRegistry.Create();
finalization
  mSomeRegistry.Free;
end.
```

Listing 2

```
unit SomeProcessor
type TSomeProcessor = class
// details omitted
end;
initialization
   GetSomeRegistry.RegisterClass(TSomeProcessor);
    // register our class
```

Listing 3

```
unit AnotherProcessor
type TAnotherProcessor = class
// details omitted
end;
implementation
initialization
  GetSomeRegistry.RegisterClass
  (TAnotherProcessor);
  // register our class
```

Listing 4

```
unit SomeProcessor
...
interface
uses
  AnotherProcessor, // <- indicate we need this
  SomeRegistry;
...
```

Listing 5

Note this happens when updating the implementation of single unit, not the program code, which remains blissfully agnostic of the changes. In this way we have been able to clearly and unambiguously capture a program dependency that was previously not knowable from inspecting the source.

There are corollaries

- RAII *per se* is out, although your classes must of course still behave sensibly
  - this may have been noticed – properties need to have workable defaults (or default behaviour that makes sense)
  - once you have committed to a property based system for configuring objects, what constructors could you possibly write? Instead of solving that hard problem, the component is plugged into the framework
- no automatic destruction of class instances
  - destruction is explicit in Delphi's Object Pascal and – key point – with the Delphi component framework the object deletion would be handled for you correctly
  - coupled with the streaming system's ability to 'automagically' find and instantiate the right classes when streaming in a definition, you spend a lot less time worrying about 'ownership' – because (a) it's done for you, and (b) if you wanted to do it yourself, you may well get it wrong or find yourself fighting the existing framework every step of the way
- classes are exclusively references types – so no objects on the stack, à la c++
  - this may feel like an intolerable constraint, but it happens to fits in well with the concept of dynamic extensibility ->

  The code to construct an object can be supplied, even updated on the fly, because *all objects are the same size* – they are the size of a pointer!
  - delivering essentially, a 'plugin' system that is capable of plugging in classes and their type metadata *on the fly*
  - by the way: the IDE does this every time you rebuild a component package you are working on in the IDE
- Exceptions can only throw objects, and also, given the singly rooted hierarchy we can always walk our way to the actual instance type if you have imported its interface
  - conveniently an extensible object designer system can simply roll back the stack from the offending starting point
  - *what is more* if the function was a property setter invoked by the IDE, then the IDE system can simply reject arbitrary failed attempts to set a property *without additional a priori knowledge of the internals of the components that are interacting*

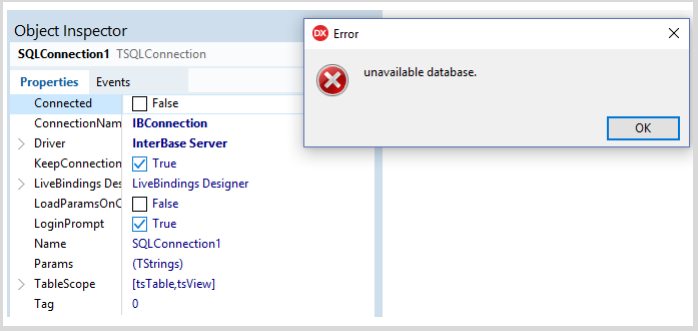


Figure 2

Example of design-time and run-time exception handling

So, let's see an example of the exception handling strategy in action. Here is the behaviour when I attempt an operation in the IDE that cannot be fulfilled (Figure 2) and how that component raised it in the code (Figure 3). Note that the same code is run in the IDE, via the component package which can be plugged in via the IDE's extensibility.

In order to investigate this I only had to add this code and hit 'Debug' – hence seeing the code by debugging the application, which will generate the same exception.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SQLConnection1.Open;
end;
```

In the IDE, the property set on the object fails, and the user is notified.

In the application, the default exception handler installed for the application is invoked, as I elected to not install my own handler, or write an exception handling block.

That's the power of a unified and usable approach to exception handling. Figure 4 shows how it looks in the app.

C'mon it can't have been that perfect, can it?

Now I have to explain why Delphi is not enjoying the popularity it once did.

Web applications

Delphi was great for software that would be popped into the post, on a CD or floppy disks.

When the web based application revolution came, that became irrelevant for many new applications. I feel the offerings within the Delphi toolbox for web development didn't seem to cut through on the feature set, and of course at the time, everything from the OS to the Development tool needed to be paid for.

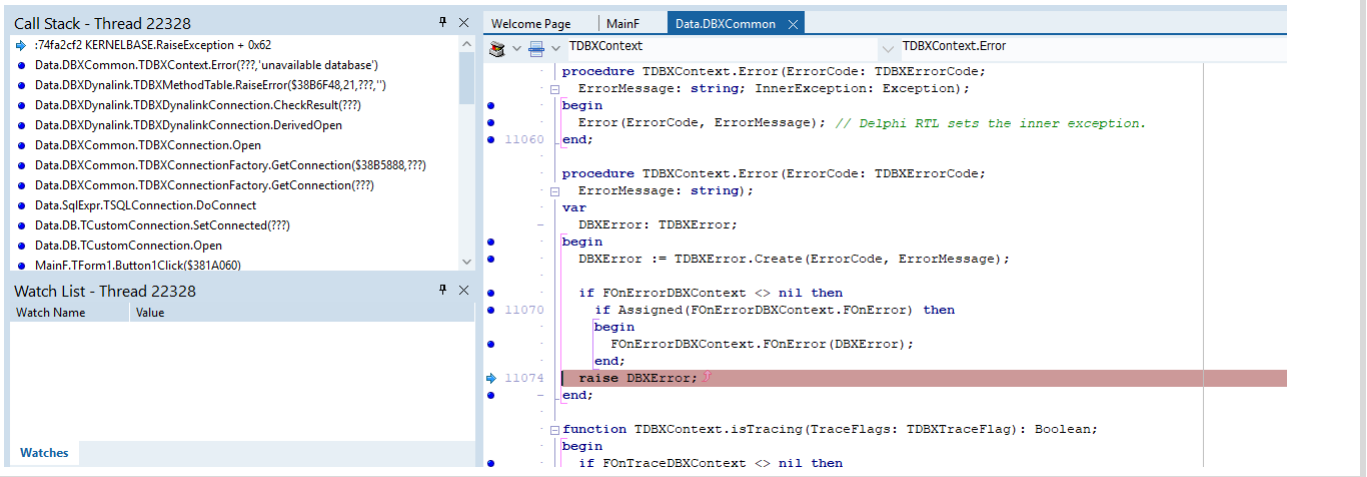


Figure 3

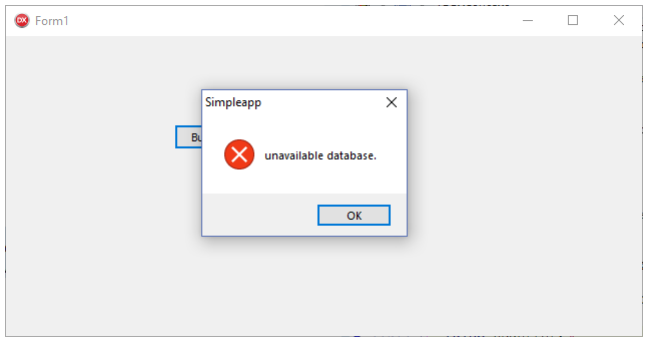


Figure 4

Given the competition at the time was the LAMP stack Linux + Apache + MySql + PHP, it was clear how that would play out.

It was proprietary

So, a fact of life is: individual companies get in trouble, go off-beam etc. this can be a real concern. For example the 64-bit compiler took a long time to appear, and some companies like to take a very long view on their enterprise applications.

Cost concerns

It could end up looking pricey compared to free tools. Yet: 'beware false economies'.

Quality issues

There were some releases that had surprisingly persistent niggles: the debugger in Delphi 4 could be a real pain, especially given the Delphi 3 debugger was an absolute pleasure to work with. This is the kind of thing that worries thoughtful programmers and managers with an eye to the future.

Another syndrome that I saw which was very sad, is that the marvellous extensible IDE was at risk from poorly programmed component packages. The IDE would be blamed for instability, when in fact, the code that it loaded into its core might well be the cause. With an improved architecture, that might have been mitigated, but perhaps not eliminated.

And finally: of course, native code is not to be 100% trusted, yet can only be run as 100% trusted.

Interfacing with code from other systems

Some might believe this was not possible, but in fact it was.

Of course the interaction with the Windows libraries was mainly via the win32 API, proving the point.

So, there was nothing preventing the user from making their own integrations, however these did require some expertise and effort to produce the translation units that could make use of the foreign function interfaces.

In fact, one of the long standing issues with Delphi for some people was that the translation units would not be updated quickly enough when new systems or features arrived in Windows. This resulted in the Delphi programmers either having to roll their own or wait for new Delphi releases.

In retrospect

So, in 2019, what conclusions can we draw?

Rapid Application Development with true visual design

Properties, Methods and Events allow complex UI to be defined in a very minimalist fashion, which is A Good Thing. There are many camps on this topic, but I hope I demonstrated above, the system supported fully visual development, all the way from specified in the designer to fully defined in code and all the waypoints between, and what is more: cleanly.

That was a strength – not all apps need to be coded the same way, or need the same level of complexity.

Strong typing can actually be fine for RAD

Caveat: with the right level of compiler and runtime co-operation.

Personally, I enjoyed debugging in Delphi, as it seemed that faults tended to be more reproducible and more easily reasoned through that some other languages.

Modules are awesome

When the programmer needs to employ a unit from another unit, they really only need to choose whether they want to add it to the interface or not – and there is a habit forming effect from the constant gentle reminder that it was preferable to factor your code well such that dependencies could be added in the implementation.

In some cases one could simply add a reference to a different module to a code file in order to modify/patch the programs' behaviour – see prior example.

How many of these concerns sound familiar even today?

I suspect we can learn much from the design precepts of the previous glory days of Delphi and take some lessons forward for the next iteration of our tools. The observant reader will spot that I mention both compile time and run-time behaviour of a feature quite often. This is uppermost in my mind because although a hypothetical rapid development environment may have well tuned strictness and guarantees in the compiler or in the serialisation system, the true art is ensuring that there is the minimum 'impedance mismatch' between those two concepts.

There is little point in polishing those systems if I then end up spending my time fighting the edge cases when they interact. Typically that borderland is where the tool support is weakest, and also, it tends to be the most user visible portion of applications. My contention is that in making that area just easier to operate in, Delphi allowed developers to focus on the parts of application development that added the most value to the user. ■

References

Working code referred to in the article can be found at https://github.com/patrickmmartin/Brute

[Embarcadero-1] https://www.embarcadero.com/products/delphi

[Embarcadero-2] https://www.embarcadero.com/products/delphi/starter

[Embarcadero-3] http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Language_Overview

[Fagg98] Adrian Fagg (1998) 'A Delphic Experience' in *Overload* 29, available at https://accu.org/index.php/journals/565

[Fahrni18] Windows GPF, published 12 August 2018 at https://iam.fahrni.me/2018/08/12/1858/

[Hague09] James Hague 'A Personal History of Compilation Speed, Part 2', available from https://prog21.dadgum.com/47.html

[OracleAtDelphi05] '10 Years of Deplphi', published on 8 February 2005 at https://blog.therealoracleatdelphi.com/2005/02/10-years-of-delphi_8.html

[Stob12] Verity Stob 'The Sons of Khan and the Pascal Spring', *The Register*, 16 January 2012, available at: https://www.theregister.co.uk/2012/01/16/verity_stob_sons_of_khan_2011/

[Wikipedia-1] 'Delphi (IDE)': https://en.wikipedia.org/wiki/Delphi_(IDE)

[Wikipedia-2] 'Object Pascal': https://en.wikipedia.org/wiki/Object_Pascal

[Wikipedia-3] 'Calling convention': https://en.wikipedia.org/wiki/Calling_convention

[Wikipedia-4] 'Component-based software engineering': https://en.wikipedia.org/wiki/Component-based_software_engineering

This article was first published on Github: https://github.com/patrickmmartin/pith-writings/blob/master/et_tu_delphi/article.md.

C++ Pipes

Expressive code can make life easier. Jonathan Boccara demonstrates fluent pipelines for collections in C++.

Pipes are small components for writing expressive code when working on collections. Pipes chain together into a pipeline that receives data from a source, operates on that data, and sends the results to a destination.

This is a header-only library, implemented in C++14.

The library is under development and subject to change. Contributions are welcome. You can also log an issue if you have a wish for enhancement or if you spot a bug.

A first example

Here is a simple example of a pipeline made of two pipes: **transform** and **filter**:

```
auto const source = std::vector<int>{0, 1, 2, 3,
  4, 5, 6, 7, 8, 9};
auto destination = std::vector<int>{};
source >>= pipes::filter([](int i)
           { return i % 2 == 0; })
       >>= pipes::transform([](int i)
           { return i * 2; })
       >>= pipes::push_back(destination);
// destination contains {0, 4, 8, 12, 16};
```

What's going on here

- Each element of **source** is sent to **filter**.
- Every time **filter** receives a piece of data, it sends its to the next pipe (here, **transform**) only if that piece of data satisfies **filter**'s predicate.
- **transform** then applies its function on the data its gets and sends the result to the next pipe (here, **pipes::push_back**).
- **pipes::push_back** **push_back**s the data it receives to its **vector** (here, **destination**).

A second example

Here is a more elaborate example with a pipeline that branches out in several directions:

```
A >>= pipes::transform(f)
  >>= pipes::filter(p)
  >>= pipes::unzip(pipes::push_back(B),
      pipes::demux(pipes::push_back(C),
      pipes::filter(q) >>= pipes::push_back(D),
      pipes::filter(r) >>= pipes::push_back(E));
```

Here, **unzip** takes the **std::pair**s or **std::tuple**s it receives and breaks them down into individual elements. It sends each element to the pipes it takes (here **pipes::push_back** and **demux**).

demux takes any number of pipes and sends the data it receives to each of them.

Since data circulates through pipes, real life pipes and plumbing provide a nice analogy (which gave its names to the library). For example, the above pipeline can be graphically represented like Figure 1 (overleaf).

Doesn't it look like ranges?

Pipes sort of look like ranges adaptors from afar, but those two libraries have very different designs.

Range views are about adapting ranges with view layers, and reading through those layers in lazy mode. Pipes are about sending pieces of data as they come along in a collection through a pipeline, and letting them land in a destination.

Ranges and pipes have overlapping components such as **transform** and **filter**. But pipes do things like ranges can't do, such as **pipes::mux**, **pipes::demux** and **pipes:unzip**, and ranges do things that pipes can't do, like infinite ranges.

It is possible to use ranges and pipes in the same expression though:

```
ranges::view::zip(dadChromosome, momChromosome)
  >>= pipes::transform(crossover) // crossover
      // takes and returns a tuple of 2 elements
  >>= pipes::unzip(pipes::push_back
        (gameteChromosome1),
      pipes::push_back(gameteChromosome2));
```

Operating on several collections

The pipes library allows to manipulate several collections at the same time, with the **pipes::mux** helper. Note that contrary to **range::view::zip**, **pipes::mux** doesn't require to use tuples.

```
auto const input1 = std::vector<int>{1, 2, 3, 4,
  5};
auto const input2 = std::vector<int>{10, 20, 30,
  40, 50};
auto results = std::vector<int>{};

pipes::mux(input1, input2)
  >>= pipes::filter   ([](int a, int b)
      { return a + b < 40; })
  >>= pipes::transform([](int a, int b)
      { return a * b; })
  >>= pipes::push_back(results);
  // results contains {10, 40, 90}
```

This article was first published on Github: https://github.com/joboccara/pipes.

Jonathan Boccara is a Principal Engineering Lead at Murex where he works on a large C++ codebase. His focus on making code more expressive. He blogs intensively on Fluent C++ and wrote the book *The Legacy Code Programmer's Toolbox*. He can be reached at jonathan@fluentcpp.com

Pipes are about sending pieces of data as they come along in a collection through a pipeline, and letting them land in a destination

Operating on all the possible combinations between several collections

`pipes::cartesian_product` takes any number of collections, and generates all the possible combinations between the elements of those collections. It sends each combination successively to the next pipe after it.

Like `pipes::mux`, `pipes::cartesian_product` doesn't use tuples but sends the values directly to the next pipe:

```
auto const inputs1 = std::vector<int>{1, 2, 3};
auto const inputs2
  = std::vector<std::string>{"up", "down"};
auto results = std::vector<std::string>{};

pipes::cartesian_product(inputs1, inputs2)
  >>= pipes::transform([](int i,
      std::string const& s)
      { return std::to_string(i) + '-' + s; })
  >>= pipes::push_back(results);
// results contains {"1-up", "1-down", "2-up",
// "2-down", "3-up", "3-down"}
```

End pipes

This library also provides end pipes, which are components that send data to a collection in an elaborate way. For example, the `map_aggregate` pipe receives `std::pair<Key, Value>`s and adds them to a map with the following rule:

- if its key is not already in the map, insert the incoming pair in the map,
- otherwise, aggregate the value of the incoming pair with the existing one in the map.

Example:

```
std::map<int, std::string> entries = { {1, "a"},
  {2, "b"}, {3, "c"}, {4, "d"} };
std::map<int, std::string> entries2 = { {2, "b"},
  {3, "c"}, {4, "d"}, {5, "e"} };
std::map<int, std::string> results;
// results is empty

entries >>= pipes::map_aggregator(results,
  concatenateStrings);
// the elements of entries have been inserted into
// results

entries2 >>= pipes::map_aggregator(results,
  concatenateStrings);
// the new elements of entries2 have been inserter
// into results, the existing ones have been
// concatenated with the new values
// results contains { {1, "a"}, {2, "bb"},
// {3, "cc"}, {4, "dd"}, {5, "e"} }
```

All components are located in the namespace `pipes`.

Easy integration with STL algorithms

All pipes can be used as output iterators of STL algorithms (see Figure 2):

```
std::set_difference(begin(setA), end(setA),
                    begin(setB), end(setB),
  transform(f) >>= filter(p)
    >>= map_aggregator(results, addValues));
```

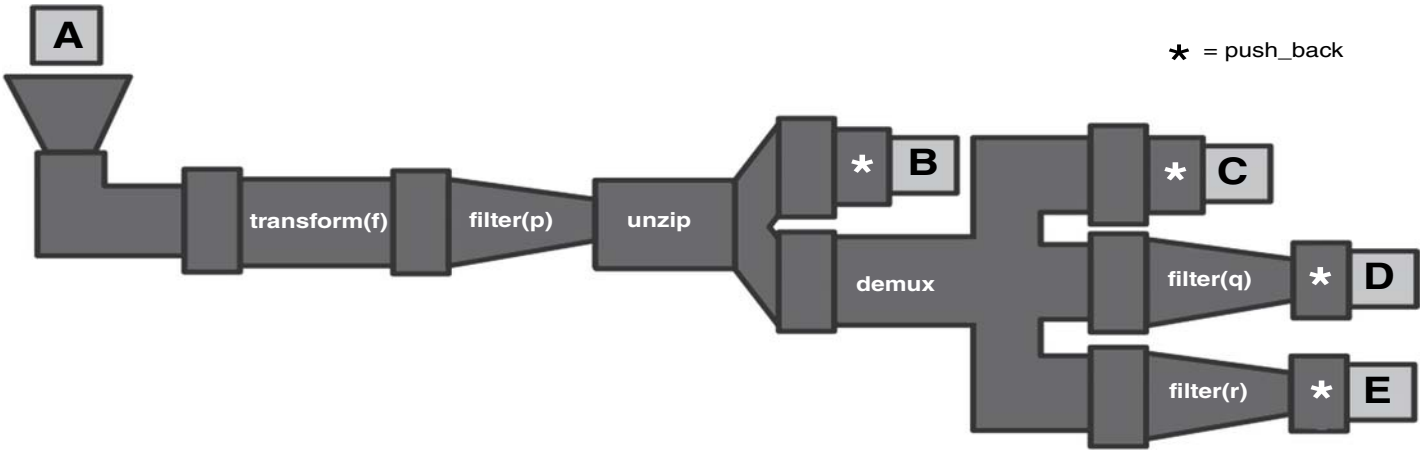


Figure 1

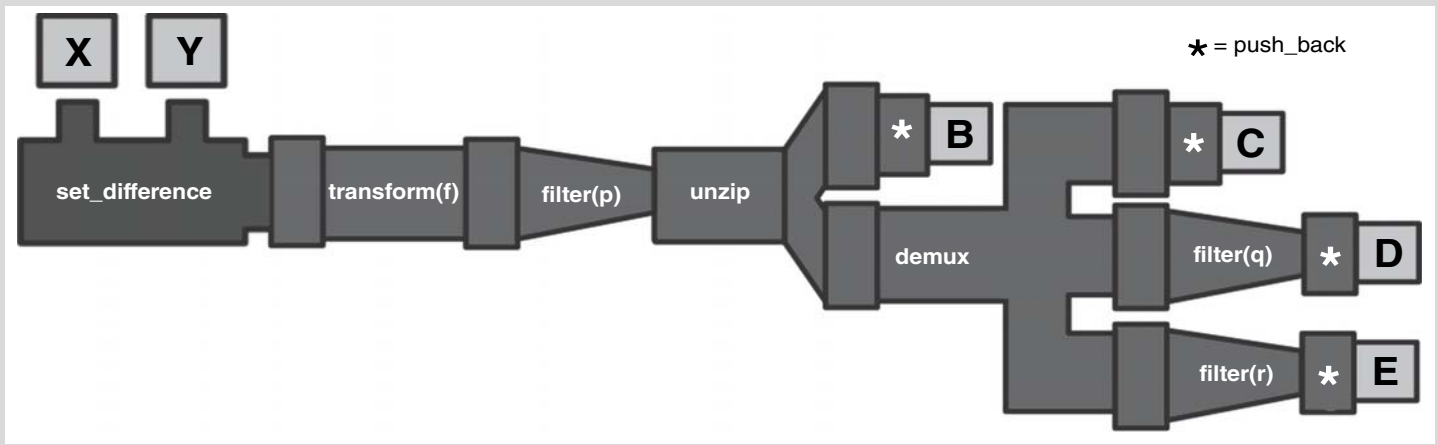


Figure 2

Streams support

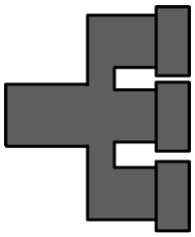
The contents of an input stream can be sent to a pipe by using **read_in_stream**. The end pipe **to_out_stream** sends data to an output stream.

The following example reads strings from the standard input, transforms them to upper case, and sends them to the standard output:

```
std::cin >>= pipes::read_in_stream<std::string>{}
         >>= pipes::transform(toUpper)
         >>= pipes::to_out_stream(std::cout);
```

General pipes

demux



demux is a pipe that takes any number of pipes, and sends a copy of the values it receives to each of those pipes.

```
std::vector<int> input = {1, 2, 3,
  4, 5};
std::vector<int> results1;
std::vector<int> results2;
std::vector<int> results3;

input >>= pipes::demux(
  pipes::push_back {results1),
  pipes::push_back(results2),
  pipes::push_back(results3));
// results1 contains {1, 2, 3, 4, 5}
// results2 contains {1, 2, 3, 4, 5}
// results3 contains {1, 2, 3, 4, 5}
```

dev_null

dev_null is a pipe that doesn't do anything with the value it receives. It is useful for selecting only some data coming out of an algorithm that has several outputs. An example of such algorithm is **set_segregate** [Boccara]:

```
std::set<int> setA = {1, 2, 3, 4, 5};
std::set<int> setB = {3, 4, 5, 6, 7};
std::vector<int> inAOnly;
std::vector<int> inBoth;
sets::set_seggregate(setA, setB,
                     pipes::push_back(inAOnly),
                     pipes::push_back(inBoth),
                     dev_null{});
// inAOnly contains {1, 2}
// inBoth contains {3, 4, 5}
```

drop

drop is a pipe that ignores the first N incoming values, and sends on the values after them to the next pipe:

```
auto const input = std::vector<int>{ 1, 2, 3, 4,
   5, 6, 7, 8, 9, 10};
auto result = std::vector<int>{};
input >>= pipes::drop(5)
      >>= pipes::push_back(result);
// result contains { 6, 7, 8, 9, 10 }
```

drop_while

drop is a pipe that ignores the incoming values until they stop satisfying a predicate, and sends on the values after them to the next pipe:

```
auto const input = std::vector<int>{ 1, 2, 3, 4,
  5, 6, 7, 8, 9, 10};
auto result = std::vector<int>{};
input >>= pipes::drop_while([](int i)
  { return i != 6; })
  >>= pipes::push_back(result);
// result contains { 6, 7, 8, 9, 10 }
```

filter

filter is a pipe that takes a predicate **p** and, when it receives a value **x**, sends the result on to the next pipe if **p(x)** is **true**.

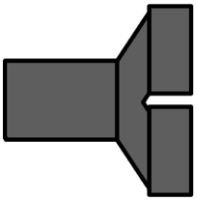
```
std::vector<int> input = {1, 2, 3, 4, 5, 6, 7, 8,
  9, 10};
std::vector<int> results;
input >>= pipes::filter([](int i)
  { return i % 2 == 0; })
  >>= pipes::push_back(results);
// results contains {2, 4, 6, 8, 10}
```

join

The **join** pipe receives collection and sends each element of each of those collections to the next pipe:

```
auto const input = std::vector<std::vector<int>>{
  {1, 2}, {3, 4}, {5, 6} };
auto results = std::vector<int>{};
input >>= pipes::join
      >>= pipes::push_back(results);
// results contain {1, 2, 3, 4, 5, 6}
```

partition



partition is a pipe that takes a predicate **p** and two other pipes. When it receives a value **x**, sends the result on to the first pipe if **p(x)** is **true**, and to the second pipe if **p(x)** is **false**.

```
std::vector<int> input = {1, 2, 3,
  4, 5, 6, 7, 8, 9, 10};
std::vector<int> evens;
std::vector<int> odds;
input >>= pipes::partition([](int n)
    { return n % 2 == 0; },
```

```
    pipes::push_back(evens),
    pipes::push_back(odds));
  // evens contains {2, 4, 6, 8, 10}
  // odds contains {1, 3, 5, 7, 9}
```

read_in_stream

read_in_stream is a template pipe that reads from an input stream. The template parameter indicates what type of data to request from the stream:

```
auto const input = std::string{"1.1 2.2 3.3"};
std::istringstream(input)
  >>= pipes::read_in_stream<double>{}
  >>= pipes::transform([](double d)
      { return d * 10; })
  >>= pipes::push_back(results);
// results contain {11, 22, 33};
```

switch

switch_ is a pipe that takes several **case_** branches. Each branch contains a predicate and a pipe. When it receives a value, it tries it successively on the predicates of each branch, and sends the value on to the pipe of the first branch where the predicate returns **true**. The **default_** branch is equivalent to one that takes a predicate that returns always **true**. Having a **default_** branch is not mandatory.

```
std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7,
  8, 9, 10};
std::vector<int> multiplesOf4;
std::vector<int> multiplesOf3;
std::vector<int> rest;
numbers >>= pipes::switch_(
  pipes::case_([](int n){ return n % 4 == 0; })
    >>= pipes::push_back(multiplesOf4),
  pipes::case_([](int n){ return n % 3 == 0; })
    >>= pipes::push_back(multiplesOf3),
  pipes::default_ >>= pipes::push_back(rest) ));
// multiplesOf4 contains {4, 8};
// multiplesOf3 contains {3, 6, 9};
// rest contains {1, 2, 5, 7, 10};
```

take

take takes a number **N** and sends to the next pipe the first **N** element that it receives. The elements after it are ignored:

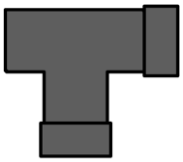
```
auto const input = std::vector<int>{1, 2, 3, 4,
  5, 6, 7, 8, 9, 10};
auto result = std::vector<int>{};
input >>= pipes::take(6)
      >>= pipes::push_back(result);
// result contains {1, 2, 3, 4, 5, 6}
```

take_while

take_while takes a predicate and sends to the next pipe the first values it receives. It stops when one of them doesn't satisfy the predicate:

```
auto const input = std::vector<int>{1, 2, 3, 4,
  5, 6, 7, 8, 9, 10};
auto result = std::vector<int>{};
input >>= pipes::take_while([](int i){
  return i != 7; })
  >>= pipes::push_back(result);
// result contains {1, 2, 3, 4, 5, 6}
```

tee



tee is a pipe that takes one other pipe, and sends a copy of the values it receives to each of these pipes before sending them on to the next pipe. Like the **tee** command on UNIX, this pipe is useful to take a peek at intermediary results.

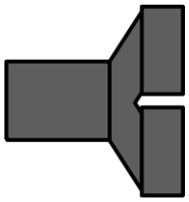
```
auto const inputs = std::vector<int>{1, 2, 3, 4,
  5, 6, 7, 8, 9, 10};
auto intermediaryResults = std::vector<int>{};
auto results = std::vector<int>{};
inputs >>= pipes::tee(pipes::push_back
  (intermediaryResults))
  >>= pipes::push_back(results);
// intermediaryResults contains {2, 4, 6, 8, 10,
// 12, 14, 16, 18, 20}
// results contains {12, 14, 16, 18, 20}
```

transform

transform is a pipe that takes a function **f** and, when it receives a value, applies **f** on it and sends the result on to the next pipe.

```
std::vector<int> input = {1, 2, 3, 4, 5};
std::vector<int> results;
input >>= pipes::transform([](int i)
  { return i*2; })
  >>= pipes::push_back(results);
// results contains {2, 4, 6, 8, 10}
```

unzip



unzip is a pipe that takes N other pipes. When it receives a **std::pair** or **std::tuple** of size N (for **std::pair** N is 2), it sends each of its components to the corresponding output pipe:

```
  std::map<int, std::string> entries
= { {1,
    "one"}, {2, "two"}, {3, "three"}, {4, "four"},
    {5, "five"} };
  std::vector<int> keys;
  std::vector<std::string> values;
  entries >>= pipes::unzip(pipes::push_back(keys),
    pipes::push_back(values)));
  // keys contains {1, 2, 3, 4, 5};
  // values contains {"one", "two", "three", "four",
  // "five"};
```

End pipes

custom

custom takes a function (or function object) that sends to the data it receives to that function. One of its usages is to give legacy code that does not use STL containers access to STL algorithms:

```
std::vector<int> input = {1, 2, 3, 4, 5, 6, 7 ,8,
  9, 10};
void legacyInsert(int number, DarkLegacyStructure
  const& thing); // this function inserts into
                 // the old non-STL container
DarkLegacyStructure legacyStructure = // ...
std::copy(begin(input), end(input),
  custom([&legacyStructure](int number){
  legacyInsert(number, legacyStructure); });
```

Read the full story about making legacy code compatible with the STL on my blog [Boccara17a].

Note that **custom** goes along with a helper function object, **do_**, that allows to perfom several actions sequentially on the output of the algorithm:

```
std::copy(begin(input), end(input),
  pipes::custom(pipes::do_([&](int i){
  results1.push_back(i*2);}).
  then_([&](int i){ results2.push_back(i+1);}).
  then_([&](int i){ results3.push_back(-i);})));
```

map_aggregator

map_aggregator provides the possibility to embark an aggregator function in the inserter iterator, so that new elements whose *key is already present in the map* can be merged with the existent (e.g. have their values added together).

```
std::vector<std::pair<int, std::string>> entries
  = { {1, "a"}, {2, "b"}, {3, "c"}, {4, "d"} };
std::vector<std::pair<int, std::string>> entries2
  = { {2, "b"}, {3, "c"}, {4, "d"}, {5, "e"} };
std::map<int, std::string> results;
std::copy(entries.begin(), entries.end(),
  map_aggregator(results, concatenateStrings));
std::copy(entries2.begin(), entries2.end(),
  map_aggregator(results, concatenateStrings));
// results contains { {1, "a"}, {2, "bb"},
// {3, "cc"}, {4, "dd"}, {5, "e"} }
```

`set_aggreagator` provides a similar functionality for aggregating elements into sets.

Read the full story about `map_aggregator` and `set_aggregator` on my blog [Boccara17b].

override

`override` is the pipe equivalent to calling `begin` on an existing collection. The data that `override` receives overrides the first element of the container, then the next, and so on:

```
std::vector<int> input = {1, 2, 3, 4, 5, 6, 7, 8,
  9, 10};
std::vector<int> results = {0, 0, 0, 0, 0, 0, 0,
  0, 0, 0};
input >>= pipes::filter([](int i)
          { return i % 2 == 0; })
      >>= pipes::override(results);
// results contains {2, 4, 6, 8, 10, 0, 0, 0, 0, 0};
```

push_back

`push_back` is a pipe that is equivalent to `std::back_inserter`. It takes a collection that has a `push_back` member function, such as a `std::vector`, and `push_back`s the values it receives into that collection.

set_aggregator

Like `map_aggregator`, but inserting/aggregating into `std::sets`. Since `std::set` values are const, this pipe erases the element and re-inserts the aggregated value into the `std::set`.

```
struct Value
{
  int i;
  std::string s;
};
bool operator==(Value const& value1,
  Value const& value2)
{
  return value1.i == value2.i && value1.s
    == value2.s;
}
bool operator<(Value const& value1,
  Value const& value2)
{
  if (value1.i < value2.i) return true;
  if (value2.i < value1.i) return false;
  return value1.s < value2.s;
}
Value concatenateValues(Value const& value1,
  Value const& value2)
{
  if (value1.i != value2.i) throw
    std::runtime_error("Incompatible values");
  return { value1.i, value1.s + value2.s };
}
int main()
{
  std::vector<Value> entries = { Value{1, "a"},
    Value{2, "b"}, Value{3, "c"}, Value{4, "d"} };
  std::vector<Value> entries2 = { Value{2, "b"},
    Value{3, "c"}, Value{4, "d"}, Value{5, "e"} };
  std::set<Value> results;
  std::copy(entries.begin(), entries.end(),
    pipes::set_aggregator(results,
    concatenateValues));
  std::copy(entries2.begin(), entries2.end(),
    pipes::set_aggregator(results,
    concatenateValues));
  // results contain { Value{1, "a"}, Value{2,
  // "bb"}, Value{3, "cc"}, Value{4, "dd"},
  // Value{5, "e"} }
}
```

sorted_inserter

In the majority of cases where it is used in algorithms, `std::inserter` forces its user to provide a position. It makes sense for un-sorted containers such as `std::vector`, but for sorted containers such as `std::set`, we end up choosing begin or end by default, which doesn't make sense:

```
std::vector<int> v = {1, 3, -4, 2, 7, 10, 8};
std::set<int> results;
std::copy(begin(v), end(v),
  std::inserter(results, end(results)));
```

`sorted_inserter` removes this constraint by making the position optional. If no hint is passed, the container is left to determine the correct position to insert:

```
std::vector<int> v = {1, 3, -4, 2, 7, 10, 8};
std::set<int> results;
std::copy(begin(v), end(v),
  sorted_inserter(results));
//results contains { -4, 1, 2, 3, 7, 8, 10 }
```

Read the full story about `sorted_inserter` on my blog [Boccara17c].

to_out_stream

`to_out_stream` takes an output stream and sends incoming to it:

```
auto const input =
  std::vector<std::string>{"word1", "word2",
  "word3"};
input >>= pipes::transform(toUpper)
      >>= pipes::to_out_stream(std::cout);
// sends "WORD1WORD2WORD3" to the standard output
```

References

[Boccara] Jonathan Boccara, 'Sets', on https://github.com/joboccara/sets

[Boccara17a] Jonathan Boccara, 'How to Use the STL With Legacy Output Collections', published 24 November 2017, available at https://www.fluentcpp.com/2017/11/24/how-to-use-the-stl-in-legacy-code/

[Boccara17b] Jonathan Boccara, 'A smart iterator for aggregating new elements with existing ones in a map or a set', published 21 March 2017, available at https://www.fluentcpp.com/2017/03/21/smart-iterator-aggregating-new-elements-existing-ones-map-set/

[Boccara17c] Jonathan Boccara, 'A smart iterator for inserting into a sorted container in C++', published 17 March 2017, available at https://www.fluentcpp.com/2017/03/17/smart-iterators-for-inserting-into-sorted-container/

Afterwood

People claim politics should not be discussed in polite circles. Chris Oldwood reconsiders.

Welcome, to the real world.
~ Morpheus (The Matrix)

Whenever I hear someone mention privileged people on TV it tends to be in reference to some upper class twit that went to one of the famous British public schools and has now found themselves in a compromising situation. The term 'privileged' is almost always used as a pejorative because it implies that the person has had access to the best education and support that money can buy and yet they have still managed to make a complete mess of things. Even if they haven't done something considered illegal, the chances are they've done something immoral or unethical instead, probably using their privileged status to gain access to the kinds of 'services' outside the reach of the general population either due to its cost, or clandestine nature.

I never went to Eton or Harrow and don't have a drop of royal blood in me, so I definitely wouldn't consider myself part of that kind of privileged society. I'm perfectly happy to snigger at their hapless mistakes and frown upon their illicit deeds as much as anyone else. But not that long ago I came down to earth with somewhat of a bump when I discovered how incredibly privileged I really am. I don't mean I suddenly discovered I was a hereditary peer or heir to a long lost rich relative, just that I realized the term 'privilege', like so many things in life, actually forms a sliding scale. It seems I haven't really been paying attention to how it affects both me and, more importantly, those I come into contact with directly, or even indirectly, through the virtual medium of social networks, forums, etc.

My cosy bubble burst when a tweet from a rather well-known (and often highly respected) member of the software engineering community got injected into my feed, which stated that politics just weren't that important. This was at a time when there was an ongoing discussion about whether it was desirable or not to keep politics out of a person's timeline that was normally reserved for more technical content. As someone who sympathized with this separation to some degree (at the time) I was somewhat intrigued to see the replies. One of the first in the thread lambasted the original poster by stating that only someone in such a privileged position as theirs could even afford to ignore politics.

And in that moment I started to comprehend where I too sat on the scale and the wave of discomfort caused me to ponder what side-effects my ambivalence might have had on others. In my naivety I would have liked to believe that my decision to eschew politics in the workplace would ensure my impartiality and therefore keep my life simple. After all, one of the many reasons to become a freelancer in the first place and remain a 'lowly' developer was to endeavour to remain near the bottom of the food chain and free of organisational concerns. Instead I'm now becoming uneasy that my choice of ignorance has in fact led me to become complicit, to some degree, through inaction. To discover someone has suffered when it might have been within my power to help is unsettling.

As a middle-aged, white, heterosexual, British male it appears to be incredibly easy to remain oblivious to what is going on to many less privileged souls, both in the workplace and in supposedly 'social' settings. By unconsciously reinforcing the patriarchy through tending to follow the overwhelmingly male technical leaders and 'influencers' you could easily be forgiven for not seeing first-hand much of the direct (and indirect) abuse and dismissive behaviour that so many other people inside (and outside) our industry suffer from. Lest you think it's implausible that someone can remain in the dark for so long, the world of programming provides the perfect escape, especially when one is lucky enough to get into it at an early age through the privilege of a computer at home. With so much interesting stuff to learn, it's all too easy to prioritise one's education efforts around the latest tech stack instead of, say, reading *The Psychology of Computer Programming*.

Hence, is it any wonder that we get seduced by the apparent logical 'utopia' of the Meritocracy when we strive for technical excellence to the detriment of our other skills? I've undoubtedly been guilty of choosing the 'best' person for a job by focusing too heavily on technical merit which has probably been reinforced by various biases that come from my own more privileged background. For example, I know in the past I've looked far more favourably on those candidates that have shown an interest in programming outside their working life.

It was well over 10 years before I met my first female programmer; none other than the editor of this very journal. Such was the status quo of working in a male dominated industry that I completely failed to notice the spelling of 'Frances' and incorrectly used the male spelling in our first few email exchanges! I might have tried downplaying this kind of faux pas in the past, as I would have other spelling mistakes, but I now see it as the result of laziness on my part – the very least I can do is pay attention and address someone correctly without making assumptions about their gender. Details matter, especially when the repercussions reinforce such an imbalance.

By making a special effort to listen to conference talks on diversity, inclusion, mental health, etc. and through following other people that freely mix their technical and personal experiences, I'm slowly turning more of my unconscious incompetence into some form of competence. The *Geek Feminism Wiki* [GeekFeminism] has been an excellent starting point, especially the topics of micro-aggressions, silencing and derailment which tend to be quite subtle in nature. Hopefully, along the way I'll try to balance out my technical and social skills somewhat more evenly so that I can use my new found position of privilege to better effect. ■

With thanks to Jez Higgins, a fellow middle-aged white dude.

Reference

[GeekFeminism] https://geekfeminism.wikia.org/wiki/Geek_Feminism_Wiki

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


accu
professionalism in programming
BOOK YOUR PLACE NOW
H & W
H & W
NEW for 2019
accu Autumn conF 2019
2 days of engaging and up-to-the-minute content plus the opportunity to listen to influencial international speakers at the forefront of software.
CONFIRMED KEYNOTE SPEAKER
Herb Sutter
www.conference.accu.org @ACCUConf
11th-12th November 2019, Belfast Hilton Hotel

intel Software

CODE MAXIMIZED

from £510

#HighPerformance

Develop high performance parallel applications from enterprise to cloud, and HPC to AI using Intel® Parallel Studio XE. Deliver fast, scalable and reliable, parallel 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 | sales@qbs.co.uk | www.qbssoftware.com/parallelstudio

qbs SOFTWARE