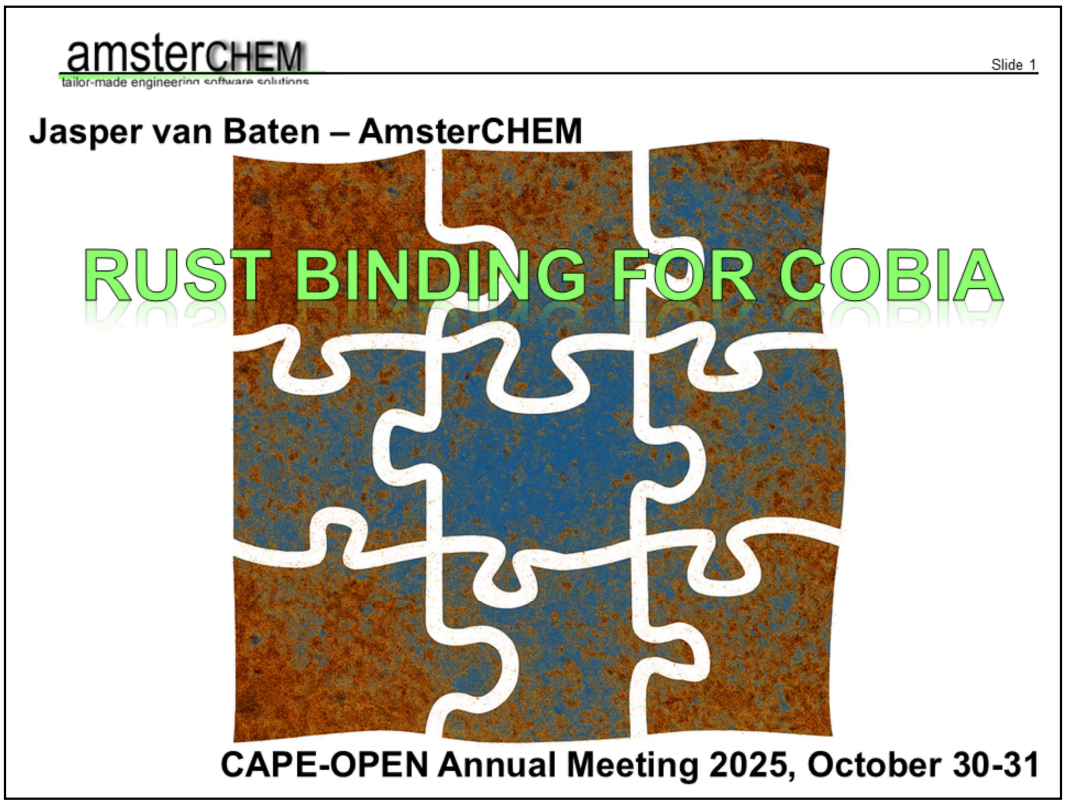
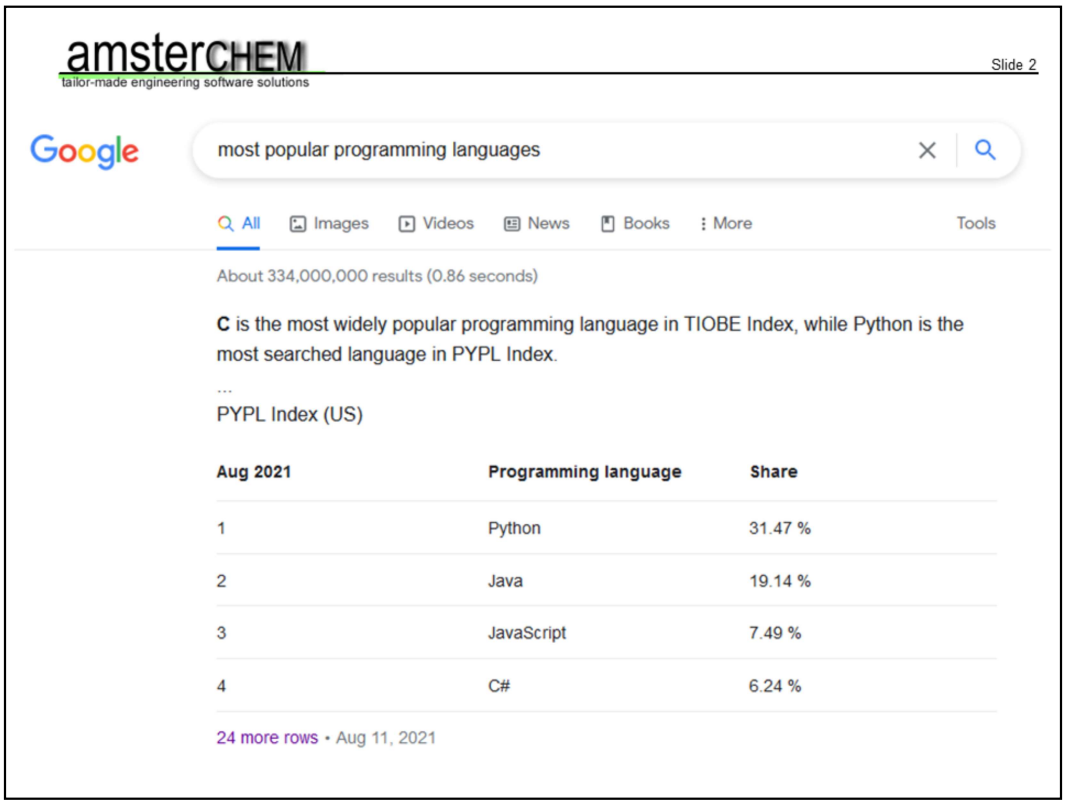
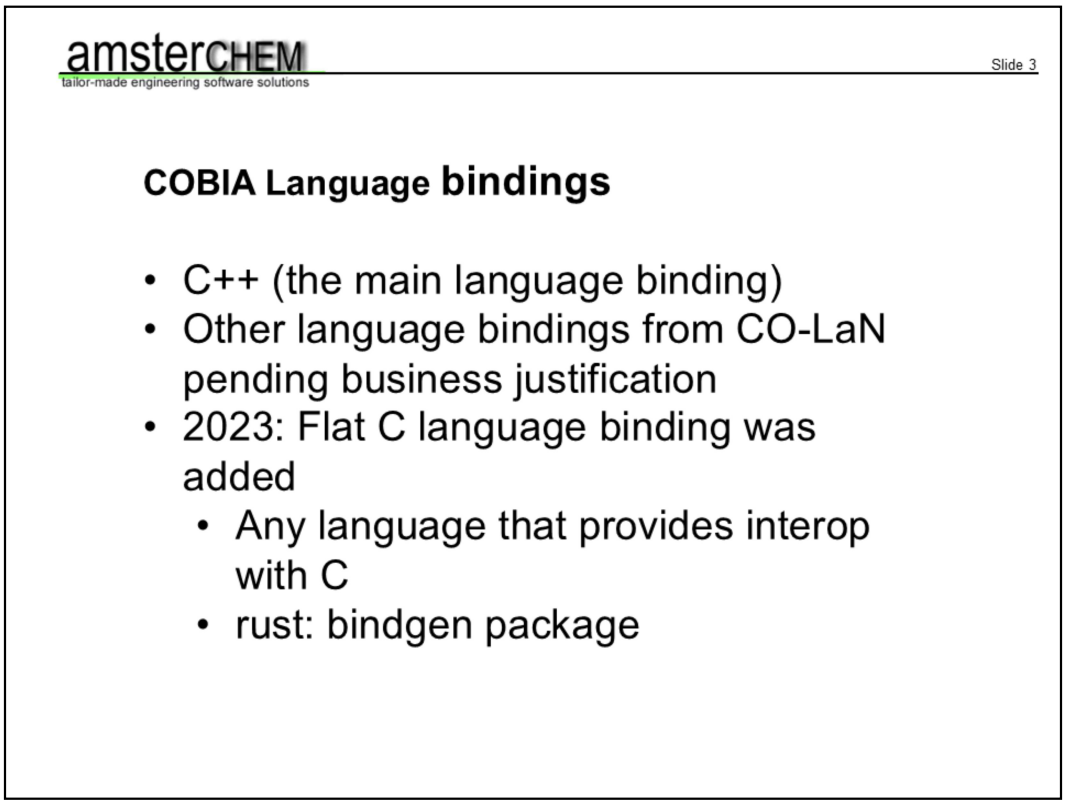


Good afternoon. My name is Jasper.

Today I will be talking about COBIA. And about RUST. And tying the two things together.



I presented the Python unit operation in 2021 – and I started with this image taken from google. This it to make the point that doing something with Python is pretty much needed at this point in time.



The Python unit operation is a COBIA implementation. It uses COBIA's C++ API. Other language bindings for COBIA are planned; however, CO-LaN does not want to commit unless there is a business justification in the form of a commitment from software vendors. This may be a bit of a chicken and egg story. So I wanted to get ahead of the game myself, and I have built up some interest in the rust language in the last couple of years. So I proposed to CO-LaN to first provide a flat C language binding to COBIA. This has manifested in 2023, and is part of the COBIA distribution itself. Of course, a C language binding can be used to in turn bind to other languages that know how to deal with the C ABI, and I then proceeded to use the C language binding for rust. One of the packages available for rust is called bindgen, which understands C.

amsterCHEM
tailor-made engineering software solutions

Slide 4

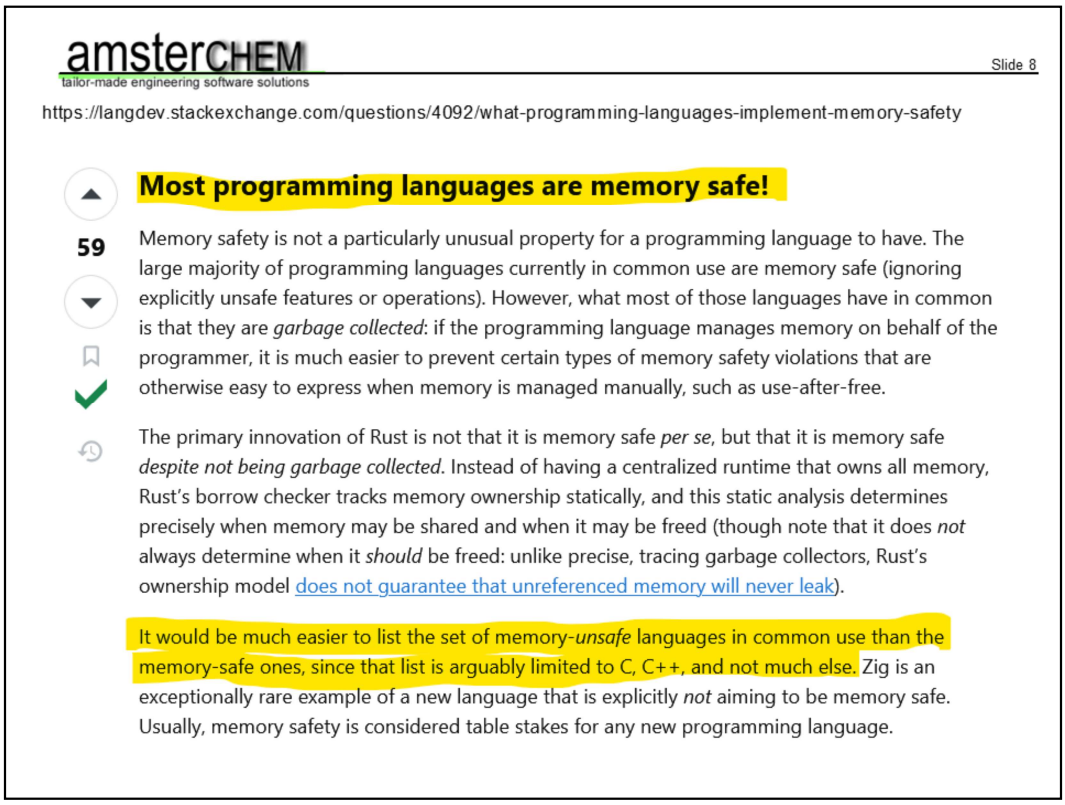
Oct 2025	Oct 2024	Change	Programming Language	Ratings	Change
1	1		Python	24.45%	+2.55%
2	4	↑	C	9.29%	+0.91%
3	2	↓	C++	8.84%	-2.77%
4	3	↓	Java	8.35%	-2.15%
5	5		C#	6.94%	+1.32%
6	6		JavaScript	3.41%	-0.13%
7	7		Visual Basic	3.22%	+0.87%
8	8		Go	1.92%	-0.10%
9	10	↑	Delphi/Object Pascal	1.86%	+0.19%
10	11	↑	SQL	1.77%	+0.13%
11	9	↓	Fortran	1.70%	-0.10%
12	29	⇈	Perl	1.66%	+1.10%
13	17	⇈	R	1.52%	+0.43%
14	15	↑	PHP	1.38%	+0.17%
15	16	↑	Assembly language	1.20%	+0.07%
16	13	↓	Rust	1.19%	-0.25%
17	12	⇊	MATLAB	1.16%	-0.32%
18	14	⇊	Scratch	1.15%	-0.26%

Looking at the TIOBE index today, we see the following picture. As an aside, interestingly C and C++ have climbed up the ladder, back to positions 2 and 3. Python still firmly holds the first place, and rust does not come in until the 16th place. So rust is not yet very popular. Then why would I invest my time in this you may wonder.

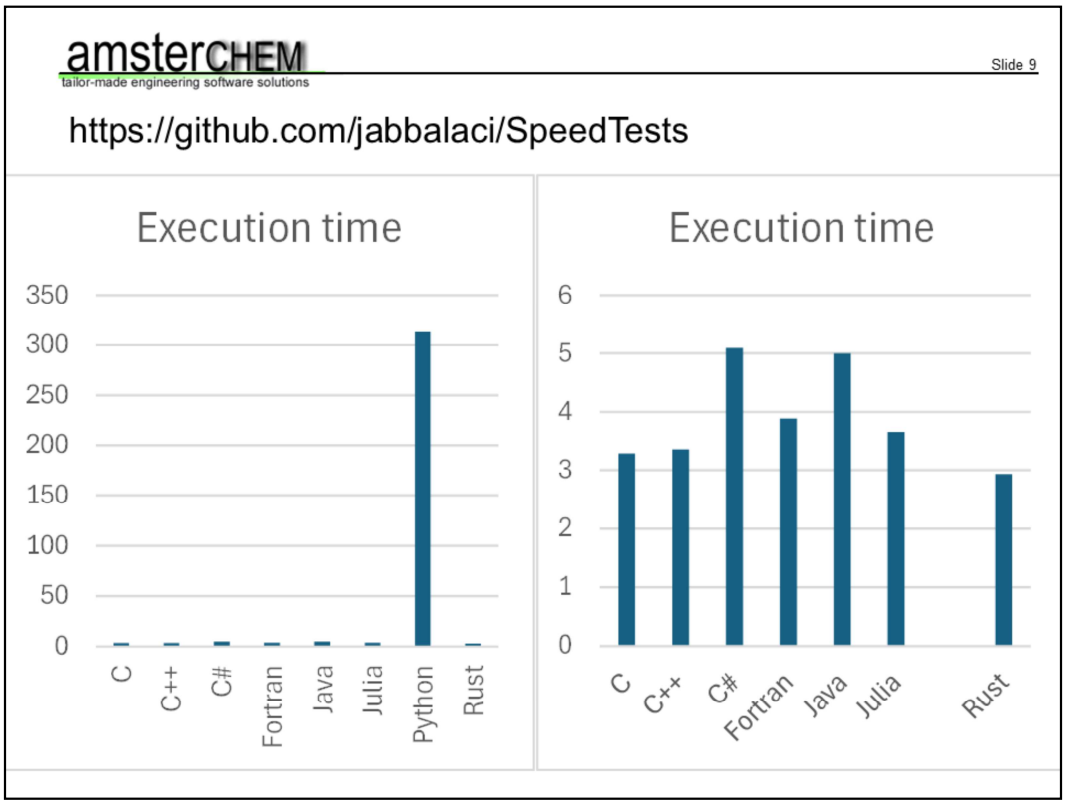
Lately we have seen quite the flood of news messages that memory overruns, dangling pointers, etc, are the bane of our existence. The United States government in particular, wants us to do something about it.

The background story here is that the memory bugs that the NSA and FBI are concerned about are the same bugs that cause security issues in software, which is in turn considered a national security risk, as these leads to exploits.

It even goes to the point where there is an FBI advisory to stop using C and C++ altogether, and use so called memory safe languages instead. Now we are not in the business of providing critical systems software typically. If we crash our simulator, it is just that, you crash the simulator. You get annoyed, restart, and complete your work anyway. CAPE-OPEN implementations have gained credible quality over the last 2 decades, so we are certainly not in a precarious situation regarding safety and stability. But it would certainly be nice to know that your software is not going to crash after releasing it.



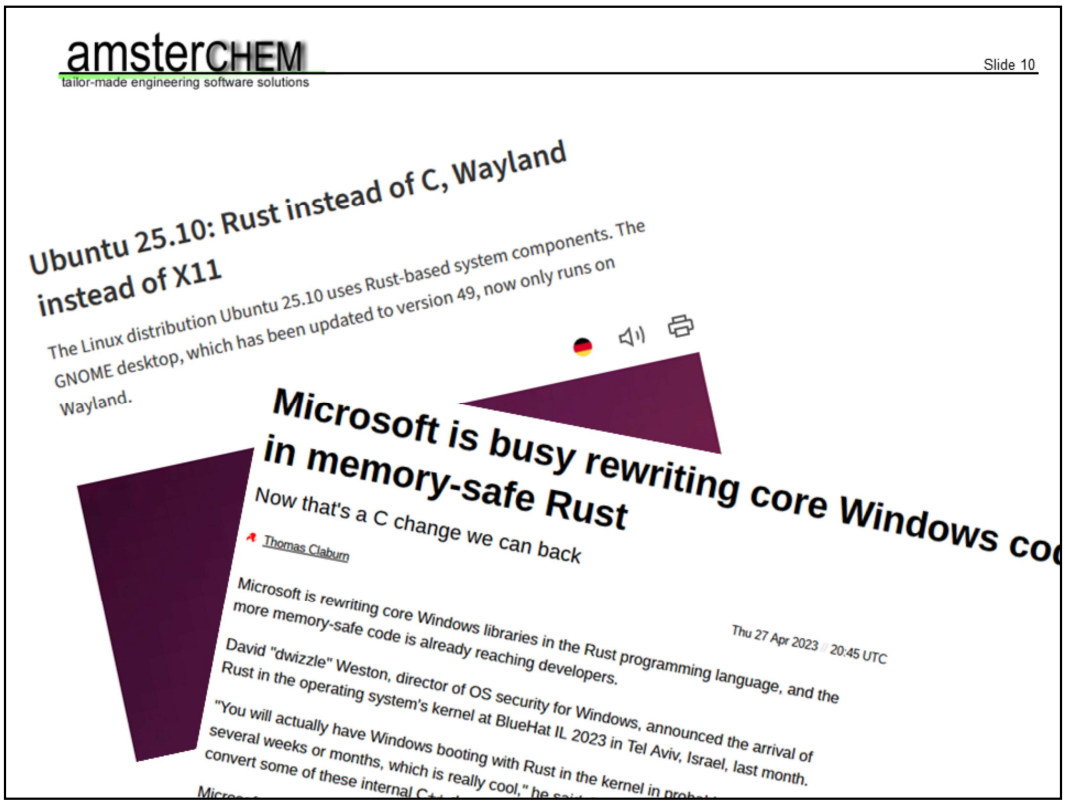
Looking further at which languages are memory safe, I found this interesting opinion at StackExchange. The author here claims that all languages, except C and C++, are essentially memory safe. He goes further in stating that the special case for rust is not in its memory safe qualities, but rather that is manages to do so without a VM and without garbage collection, but with reference counting on its allocated objects. So now we get into the realm of performance, which, to us simulation people is of course rather important.



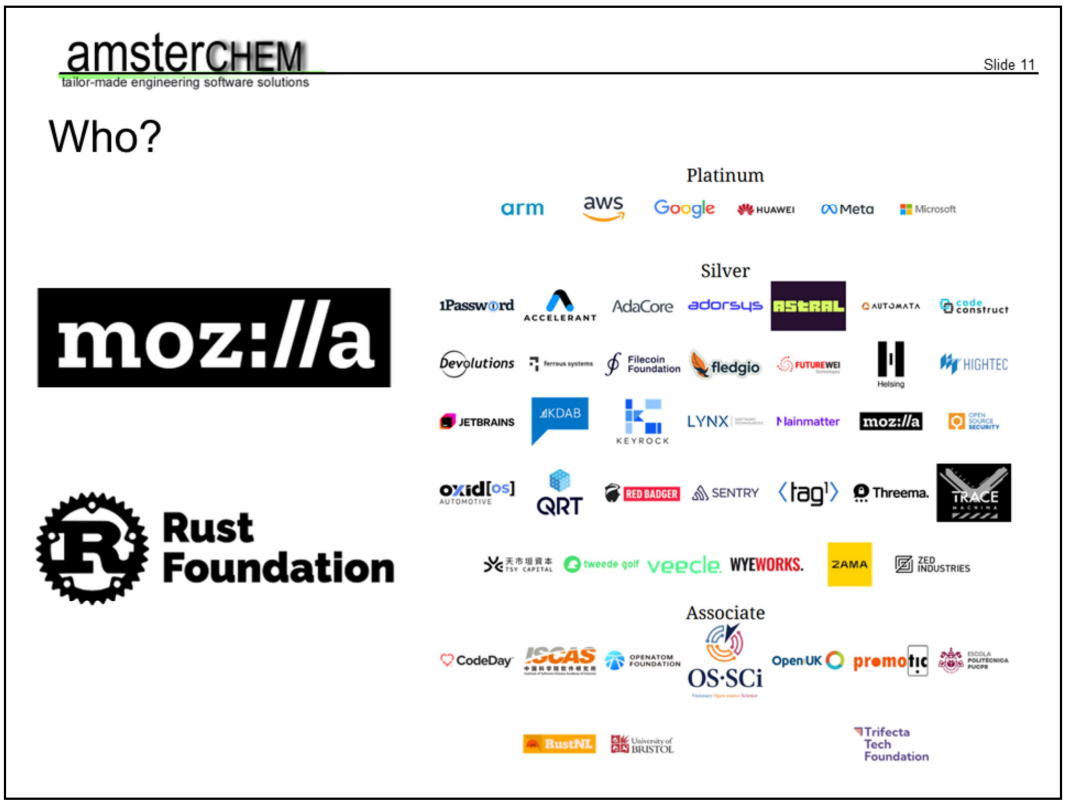
Looking up some benchmark software in which various languages are compared, we quickly get the picture. Here's a hand-full of popular languages with their performance. The first graph is somewhat obfuscated as I put our mighty popular Python in there, which is exceedingly non-performant. Not good for the CO2 production. But of course, such a statement is not entirely fair, as most Python based software uses Python merely to steer objects that are themselves written in other languages, mostly C and C++, but then again, you could claim that doing this you are not really memory safe to begin with.

On the right, Python is removed from the image and the true picture emerges. C and C++ are nearly unparalleled in performance. The numbers shown here for C and C++ are averages of different compilers with different optimizer settings; the message being here that you can get slightly better performance. Low and behold, rust is the only language that competes with that.

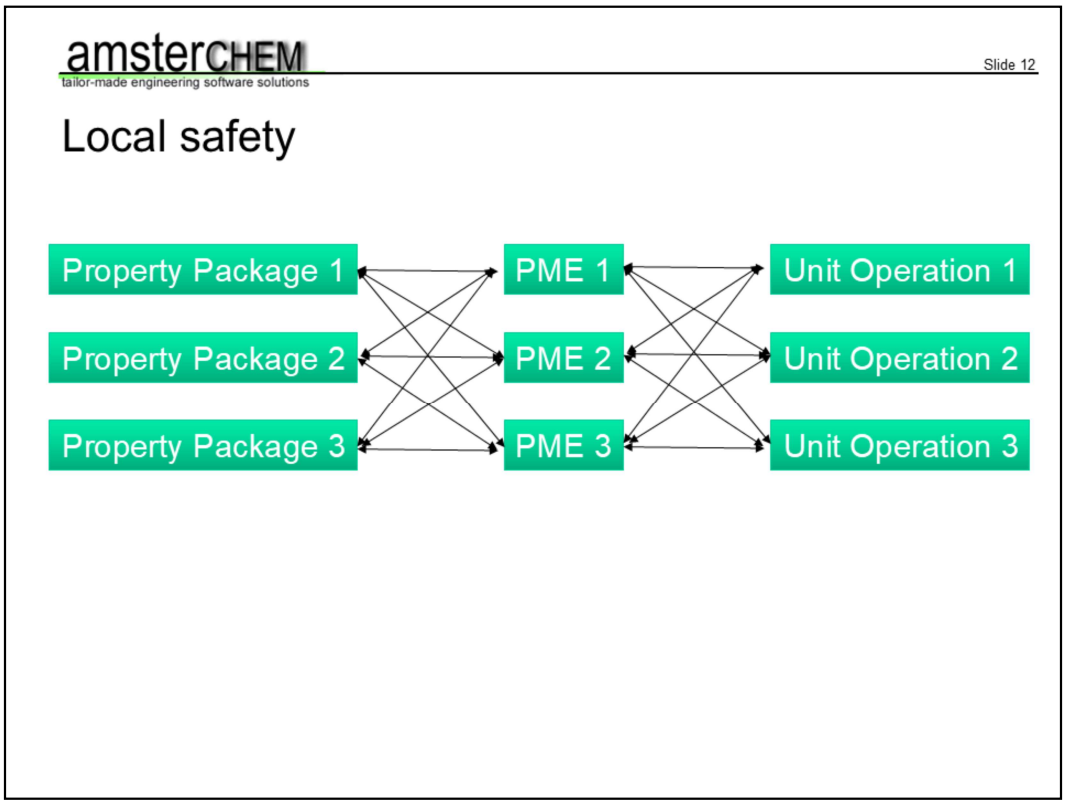
Now benchmarks being benchmarks, this is just an example and true milage may vary. But fact remains that rust is designed for performance. So what you get is performance, and safety, hand in hand.



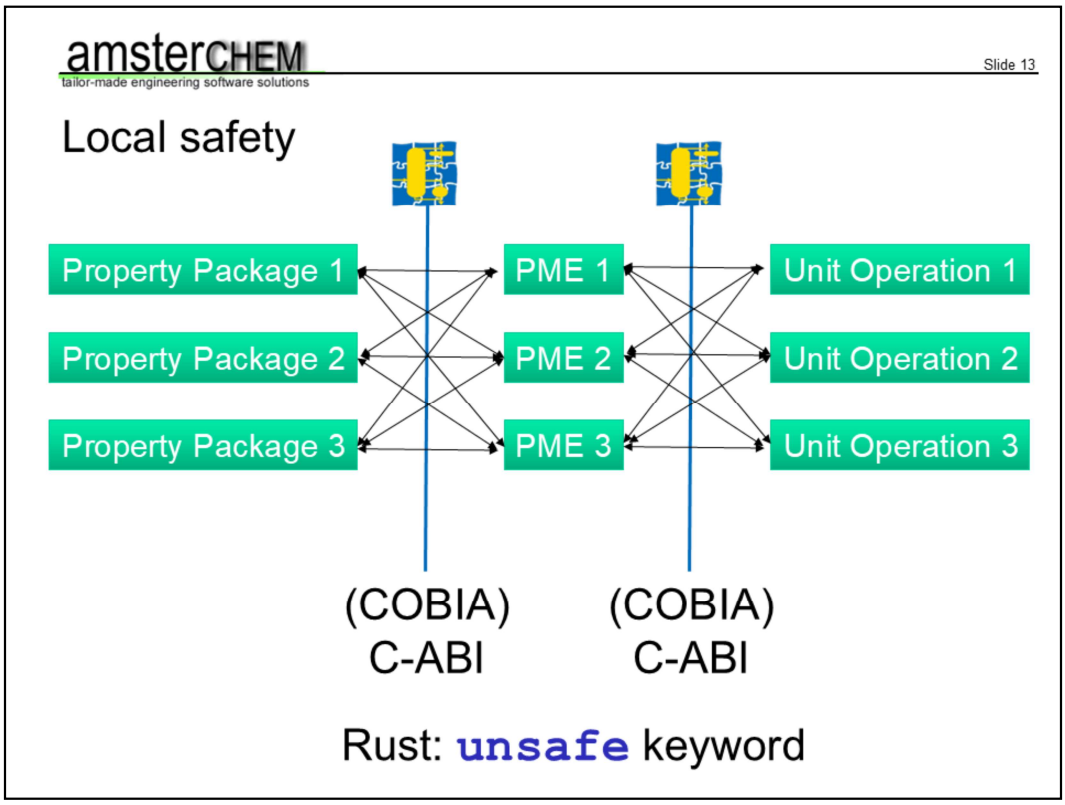
You can take my word for it. Or you can take the word of Microsoft and Linus. System utilities in both Windows and linux are being rewritten in Rust.



Languages come and go over time, and what was popular last year may be extinct a few years later. We all used Visual Basic 6 at some point. But I would think rust is here to stay. It started at Mozilla, and moved on to the rust foundation. Which is backed by some pretty big names, including Amazon, Google, and Microsoft. Rust is something I would trust to bet on at this point. And I did.



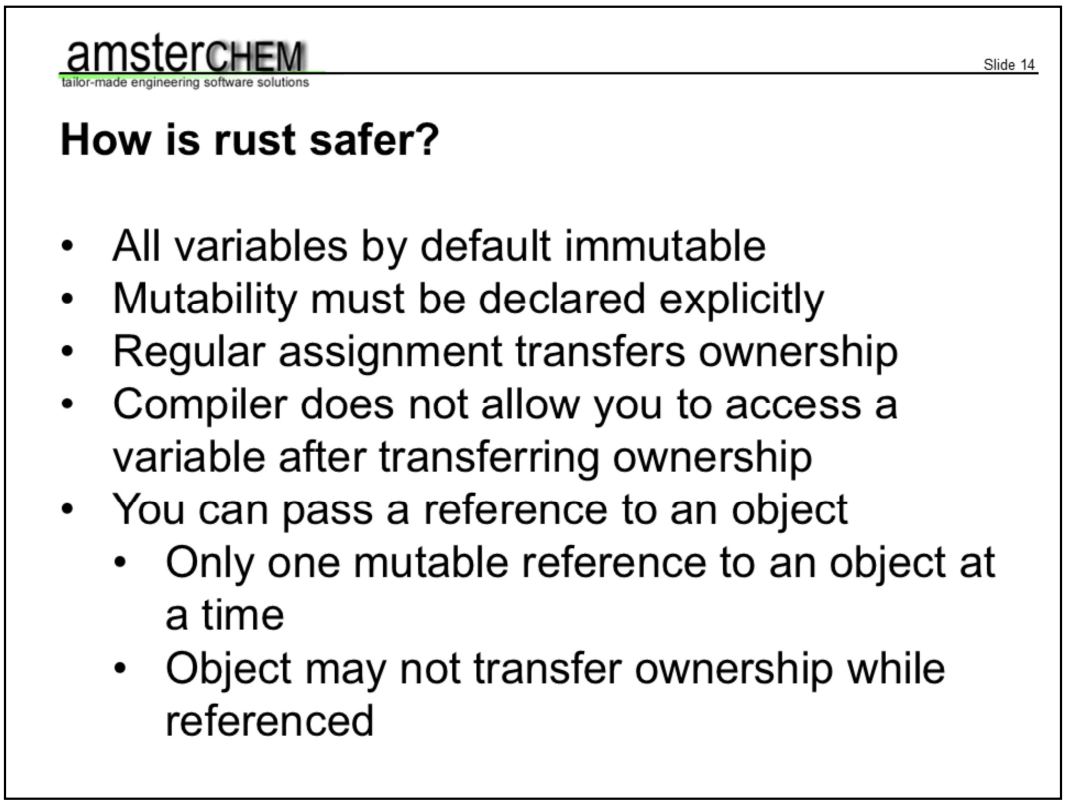
Of course, when we speak about CAPE-OPEN we are talking about interoperability. Here's the general picture. You can have multiple property package implementations, multiple process simulators and multiple unit operations, and they all talk to each other using the same language. They are not all going to be written in rust. This is also not a requirement that we want to make on the CAPE-OPEN standard.



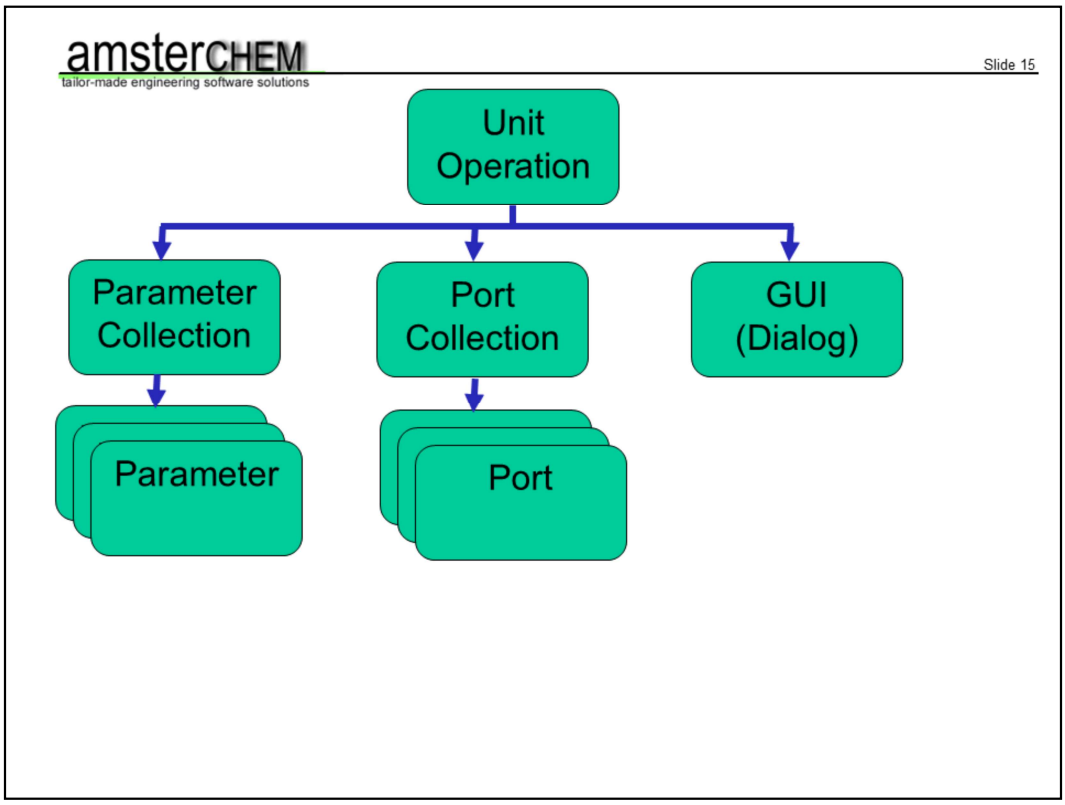
The CAPE-OPEN standard lives on the interface between these applications; CAPE-OPEN is the layer over which these applications talk to each other. COM has its own Application Binary Interface, and COBIA was designed around the C ABI, which is well defined on all platforms. So your rust components will talk to other, non-rust components, and even if you would only have rust based components in your particular setup, over the pipeline we still use C pointers to data, raw memory, etc. This is good, as this allows interop. Rust can of course do this to, with its 'unsafe' keyword. Which you will therefore find around each function that converts rust code to the CAPE-OPEN piple line and vice versa. This is not something bad, this `unsafe` keyword just alerts the programmer that the rust compiler will not have your back when it comes to safety and you need to pay attention. In particular, at this boundary you need to pay attention to the CAPE-OPEN standard and the contractual agreement that comes with that.

So now that I mentioned that we are not writing systems software where security is paramount, and we cannot avoid talking to other non-rust components, and we must in any case use the C-ABI over the CAPE-OPEN pipeline, you could wonder why to go this route at all. Particularly as rust is not an easy programming language. You will probably spend more time programming the same thing in rust as compared to other languages as the compiler it outright pedantic. But there is a clear trade off here. All the bugs that the compiler will not allow you to introduce, will not end up in your production software, and will not bother your customers. Also the extra effort spent on coding, you probably get back on not having to troubleshoot bugs

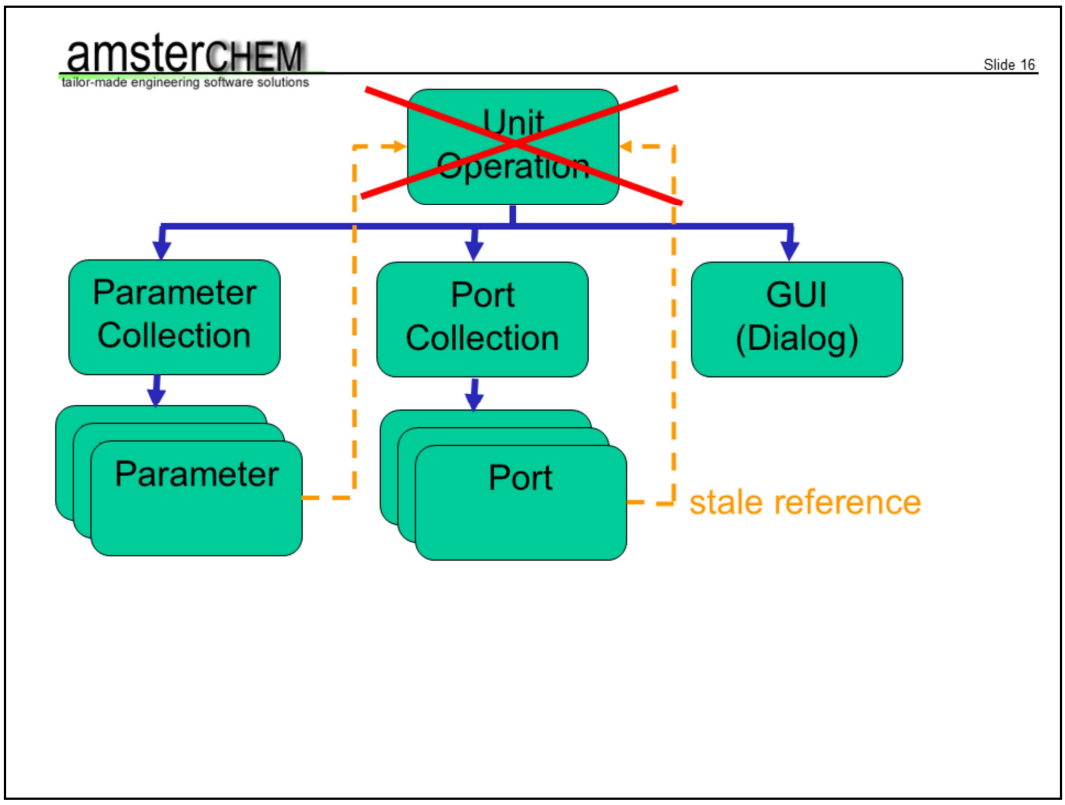
that are in your production software.



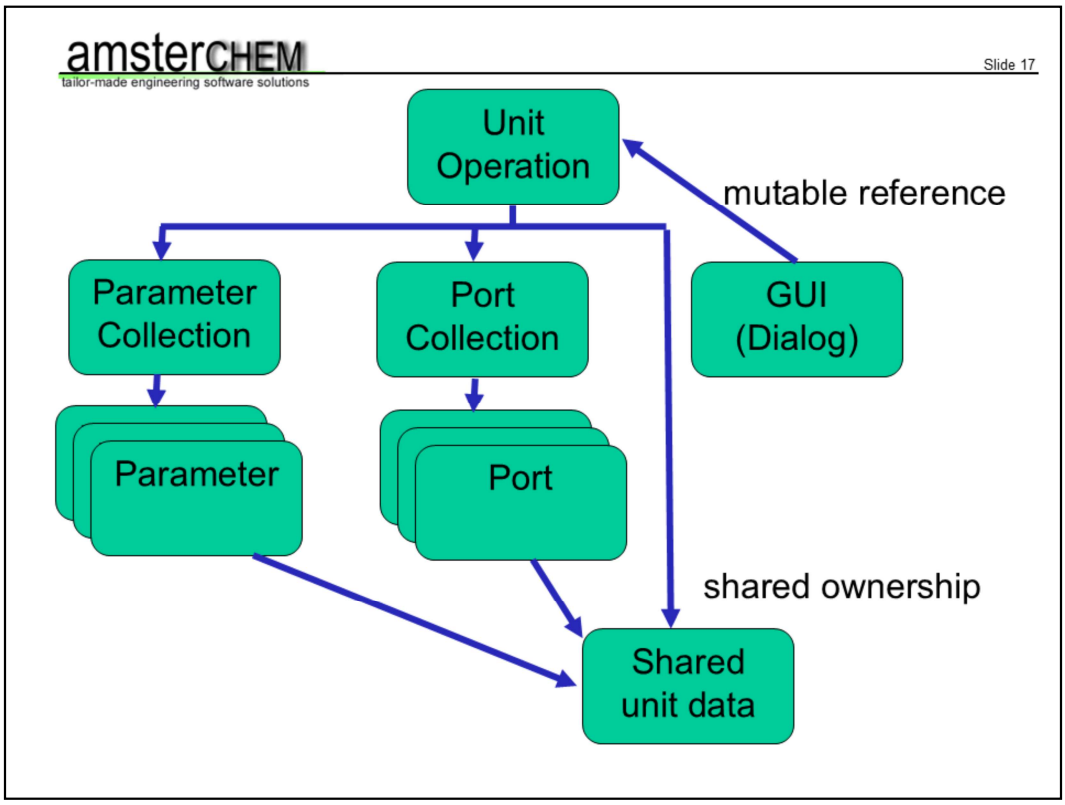
Here's essentially why rust coding is harder, and why the compiler produces safer code. The compiler will enforce you to set up your design in such a way that a whole category of potential problems is detected and disallowed at compilation time.



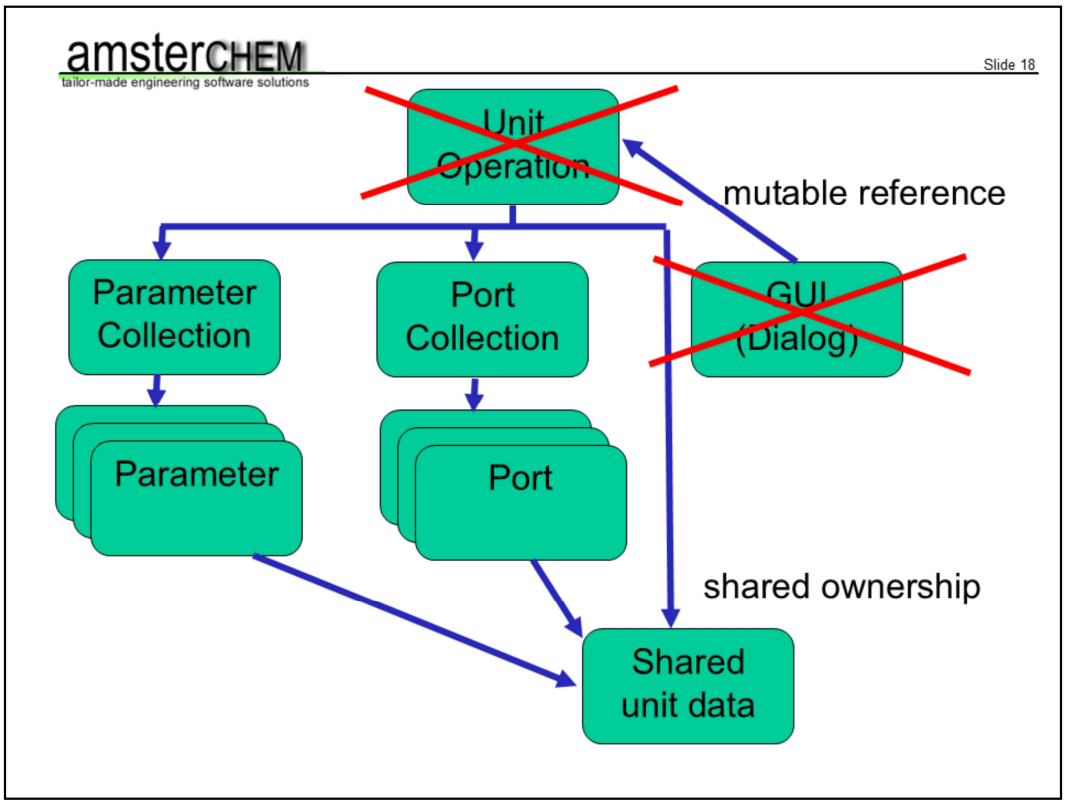
This of course requires you to rethink some of your software design. The sketch here represents how we intuitively think of a unit operation class organization, and this is also how it is often implemented. By myself at least. There is one top level unit operation class, which owns a parameter collection, which in turn owns a bunch of parameters. There is also a port collection which is owned by the unit operation, which owns a bunch of ports. And there is a dialog to edit the unit, also something that feels like it belongs to the unit operation. But, when you connect a port, you need to invalidate the unit operation, so the port must somehow know how to operate on the unit operation itself. When you change a parameter, same thing, and also you must mark the unit dirty. And you may need to access the name of the unit operation in error messages: “the material object connected to Feed port of unit operation Separator1 does not contain temperature”. So all of these owned objects must somehow access the object that owns them. Rust does not allow for this. Not easily at least.



As the unit operation is exposed to the outside world, and so are the parameters, there is no guarantee that the ports do not survive the unit operation (well – there is the CAPE-OPEN contract that says you must at least disconnect the ports at Terminate). So what happens if the Unit itself is destroyed and the ports or parameter live on? You have yourself a dangling pointer reference. This is one of the memory errors that rust aims to prevent.

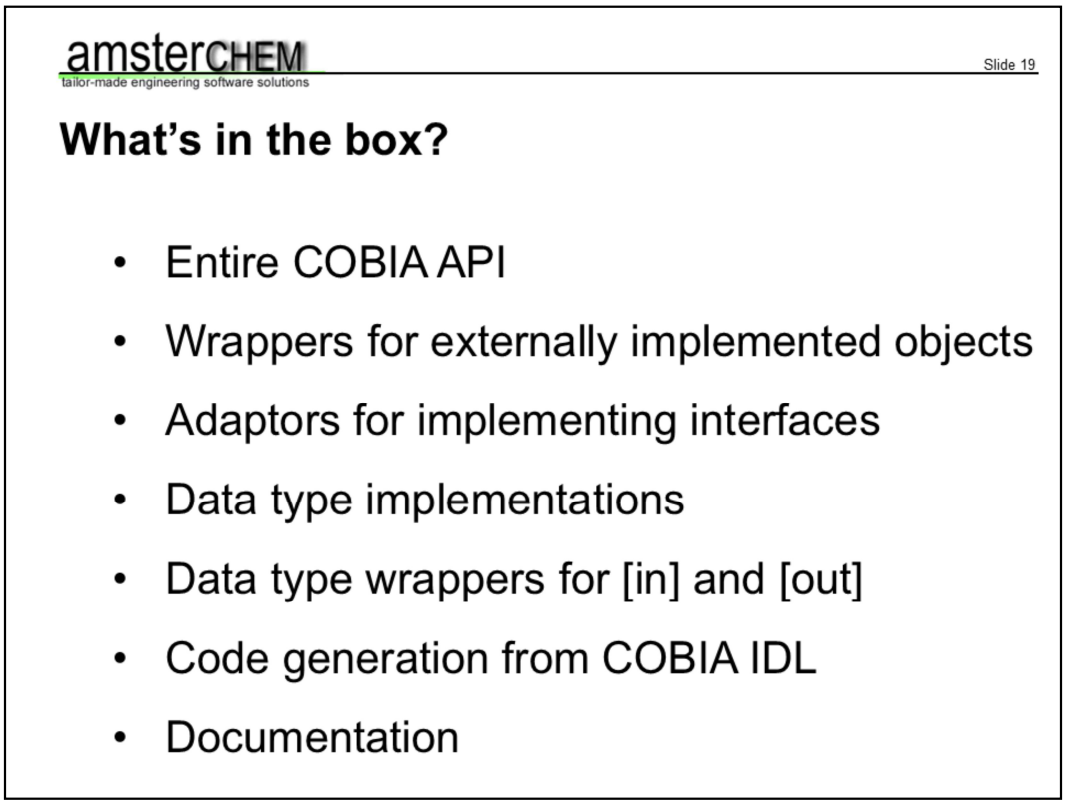


So you rethink your strategy. You could obtain the same structure as in the previous slides, using weak pointer constructs, but here's another way to look at it. Suppose the validation states, the dirty flag, the unit operation name are not actually members of the unit operation, but live in a separate object that represents the unit operation's shared data. Now the unit operation can own this data, but so can all ports and parameters. Shared, reference counted, ownership is a straight forward rust concept.

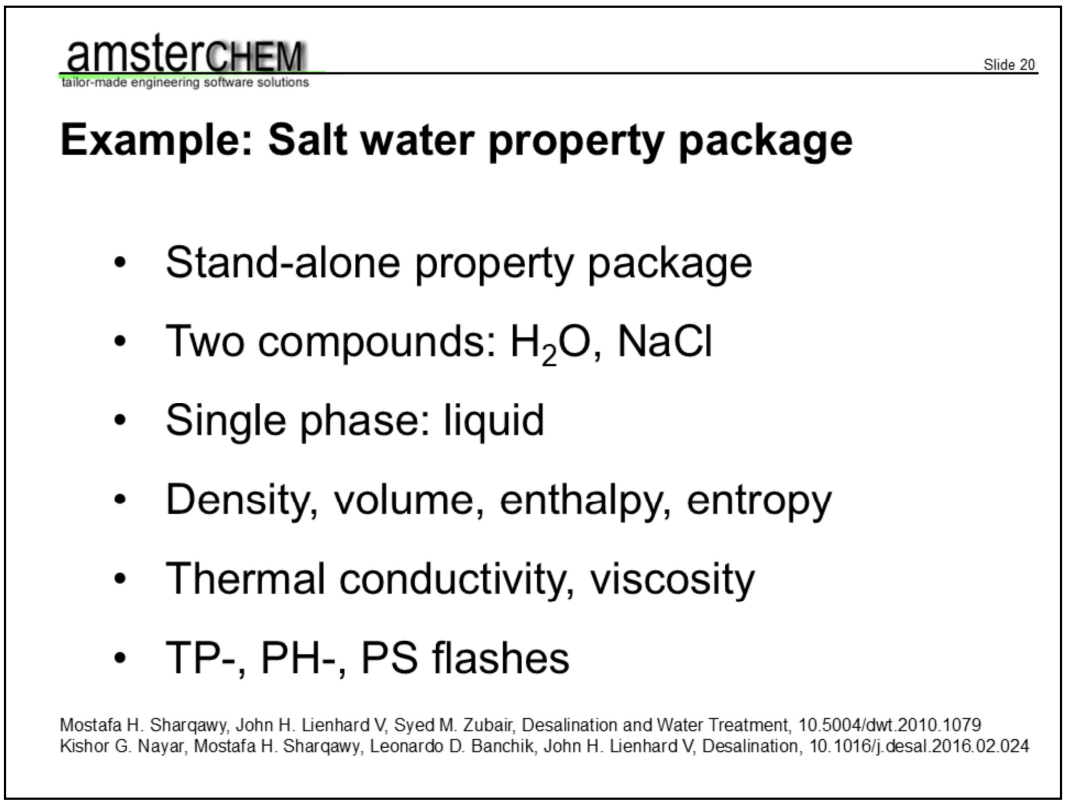


Now if the unit operation gets destroyed, there are no dangling pointers, and the shared unit data remains alive as long as any of its owners refer to it. Problem solved. The GUI construct in the previous slide is actually backwards. The unit should not own the GUI, but the GUI should own the unit, or at least a mutable reference to it. Remember from a previous slide that as long as the GUI has a mutable reference to the unit, nobody else can have a have a mutable reference to the unit. But this is entirely in line with the CAPE-OPEN contract as the GUI is modal, and the entire application grinds to a halt until the GUI is done.

Clearly you can also do all these same changes in a C++ based implementation. But the difference is that the rust compiler forces you to think about it and do it in a safe way, whereas in C++ you are free to do it safely, but you can also do it unsafely if you so wish and potentially end up with your dangling pointers.



Ok – enough tech talk. Let's see what we have. The rust cobia language binding contains a rust wrap of the entire COBIA API (well – nearly entirely at this point, I still need to dot the i's on some of it). This includes registry access, object instantiation, string utilities and other API functions. And, much like the C++ language binding, the rust cobia crate provides wrapper for externally implemented objects, adaptors for implementing your own objects in rust where the nitty gritty C details get hidden from the programmer. The crate provide some default data type implementations for the COBIA data interfaces. I decided to make [in] and [out] specific versions of the data types, as this corresponds to the rust concepts of mutable and immutable, remember that the compiler forbids multiple mutable references to the same object, but allows multiple immutable references. And it comes with a code generator that takes COBIA IDL and produces all of the above. This is in fact how the entire CAPE-OPEN 1.2 type library is converted to rust. The rust language comes with an excellent documentation system, with as bonus that any example code simultaneously servers as unit test code. All of that is included.



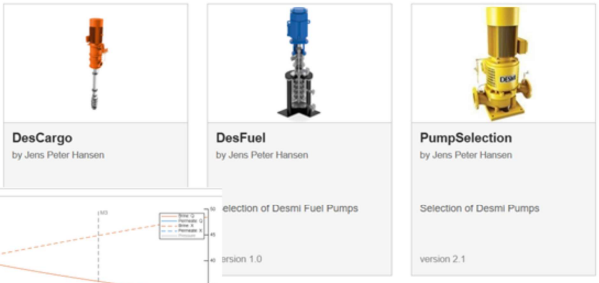
The cobia crate is part of a bigger project which is available publicly from github. It includes two examples at this point. The first example is a salt water stand-alone property package. Two compounds: salt and water. One phase, liquid. Single phase properties as well as transport properties are provided, along with a TP-, PH- and PS-flash.

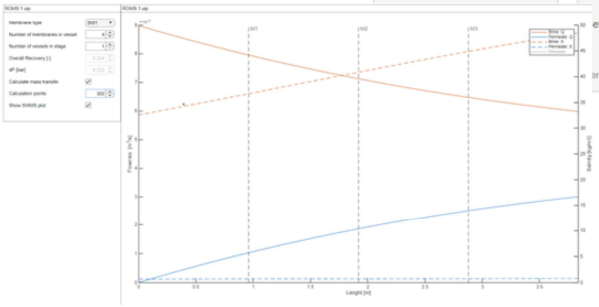
amsterCHEM
tailor-made engineering software solutions

Slide 21

Example: Salt water property package

Pump design web application





RO simulator

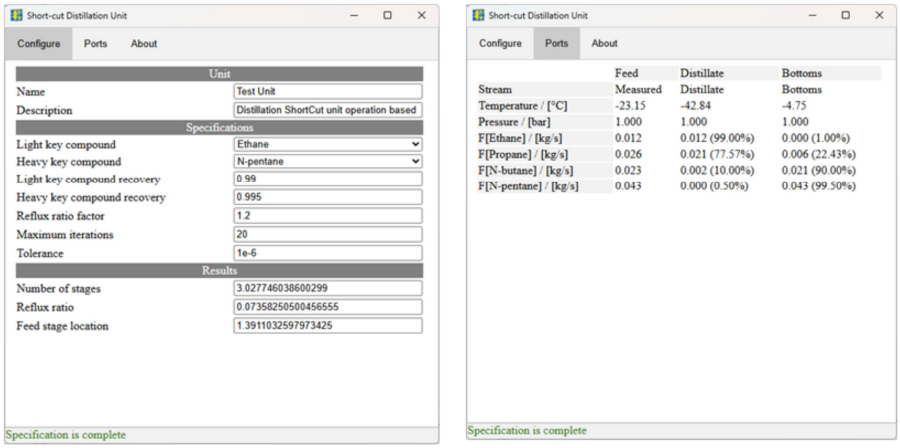
https://www.colan.org/presentation/co2023-eca-ccs/

To demonstrate that for this project I am looking for usable examples, the salt water package is already in production. You may remember Jens Peter Hansen's presentation on CAPE-OPEN use in ECA Engineering Aps, from 2023. The link is at this slide. He shared that the package is already applied in some web applications of the products shown on this slide, and in their reverse osmosis simulator, and he has plans in his company to make further use of the package.

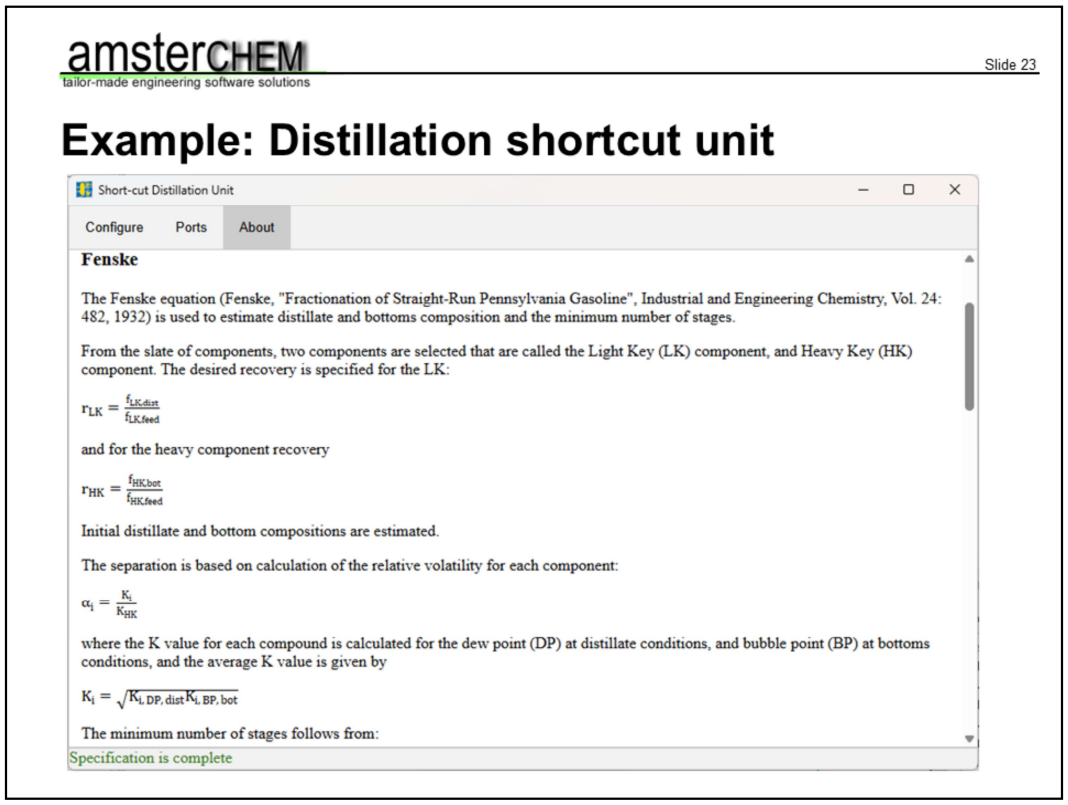
amsterCHEM
tailor-made engineering software solutions

Example: Distillation shortcut unit

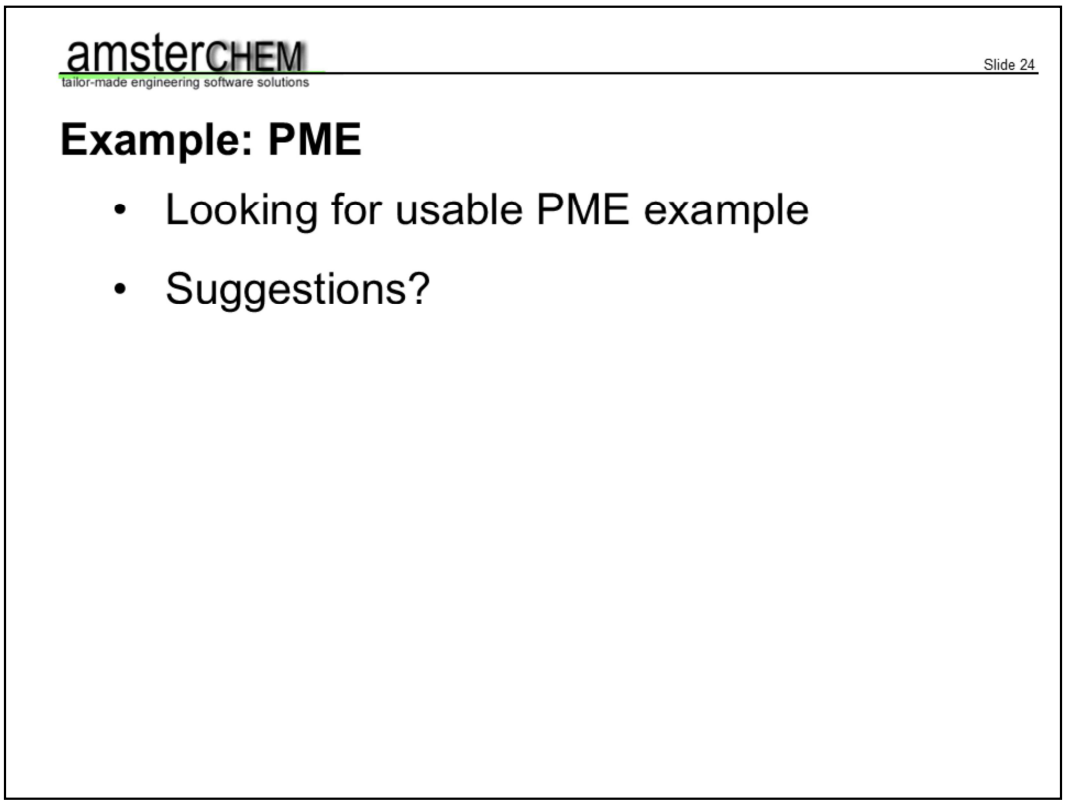
- Fenske-Underwood-Gilliland-Kirkbride
- Feed, Distillate & Bottoms ports



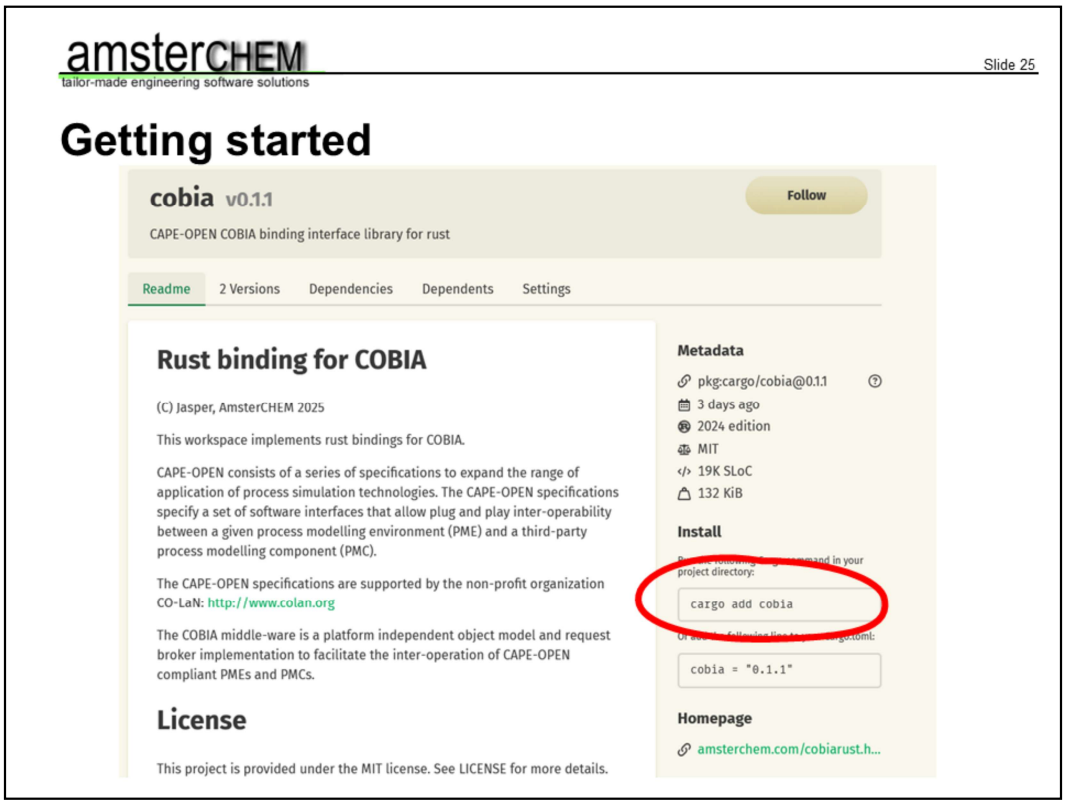
The second example is a shortcut distillation unit based on the work of Fenske, Underwood, Gilliland and Kirkbride. Shown here is the private GUI of the unit operation, showing the public parameters and a summary of the stream table, including component recoveries at the product ports.



The unit operation GUI is actually a web browser, that runs on the main GUI thread along with a web server that provides the interaction with the unit operation. As much as rust is getting mature, a one-size-fits-all widget toolkit is not yet available to my taste do I decided to go the way of the embedded web browser. This may eventually be in any case what we all end up doing if indeed everything moves to the cloud. What you are looking at here is a WebView2 browser on windows running inside a native window. I have not yet implemented the web bit on linux, so the unit compiles and runs fine on linux, but without its edit capabilities. I need to find some time to finish that bit.

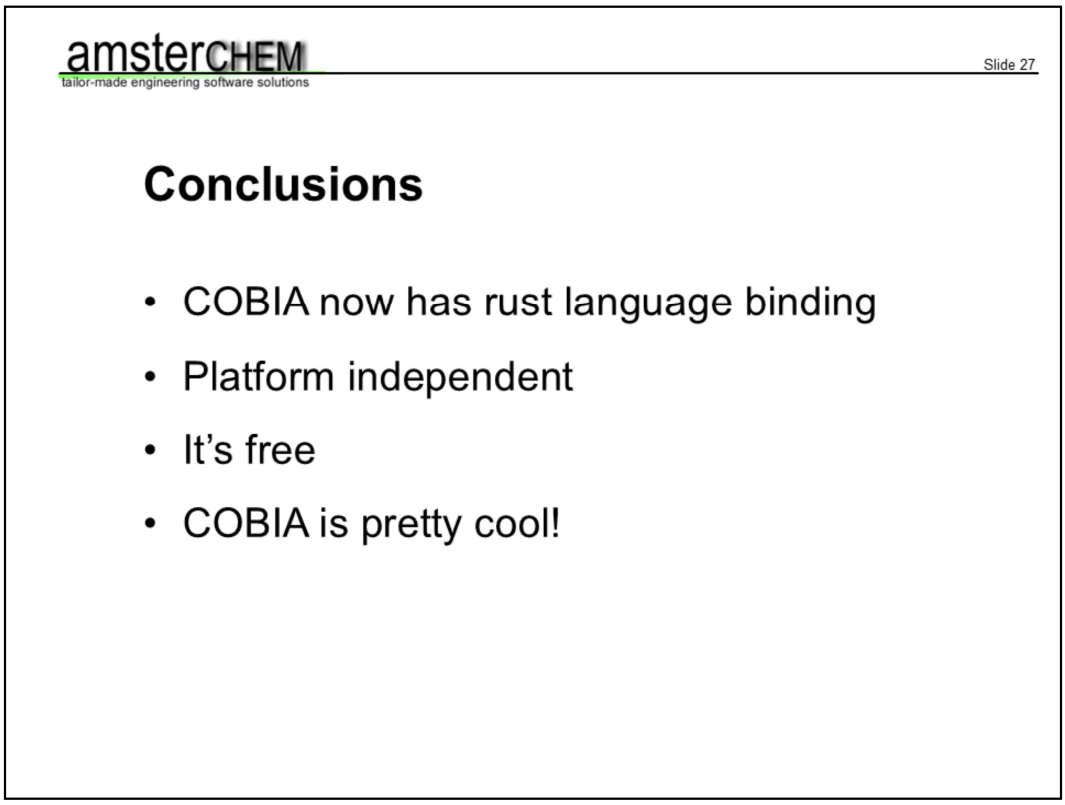


Of course, it would be nice to have a PME example as well that is not just another command-line based mixer/splitter implementation. To get the best bang for the buck, I was looking to implement examples that do not only demonstrate how to use CAPE-OPEN and particularly how to use it in the rust context, but it would also be nice if the examples have a right of existence of themselves as useful application. If anybody has a good idea on what I should implement as example PME, I am open to suggestions and I am hoping for your feedback on this item.



Rust features its own building tools and packaging tools, and independent libraries such as the cobia binding are distributed in the form of crates. As such, the rust cobia binding is now available on crates.io, where you can read that, to add this to your own rust development, all you need to do is run rust's packaging tool cargo, with the words add cobia. And you are off. Well – almost.

For it to compile, you will need to have the COBIA SDK installed, as that is how the crates learns about the COBIA API content and about the CAPE-OPEN types. It literally compiles all of this on the fly. For that in turn, you need to satisfy some prerequisite requirements of the bindgen package, most noticeably, you need to have a CLANG compiler installed. All of this is available under the do-as-you-want MIT license. Just don't hold me responsible for your damage or loss of mental sanity.



Well – I hope you enjoyed that. I did. To conclude: COBIA now has a rust language binding, which is based on COBIA's C language binding that was recently added. You make a PMC this way, and as a bonus you get that it will run on Windows and linux, as soon as we publish COBIA there. If you are CO-LaN member you can already run COBIA on linux, but you have to compile it yourself from the code. It's free – I hope you find a good use for it. And I cannot escape reaching the same conclusion as all previous years, which is that COBIA is pretty cool. Thank you for your attention.