amsterCHEM
tailor-made engineering software solutions

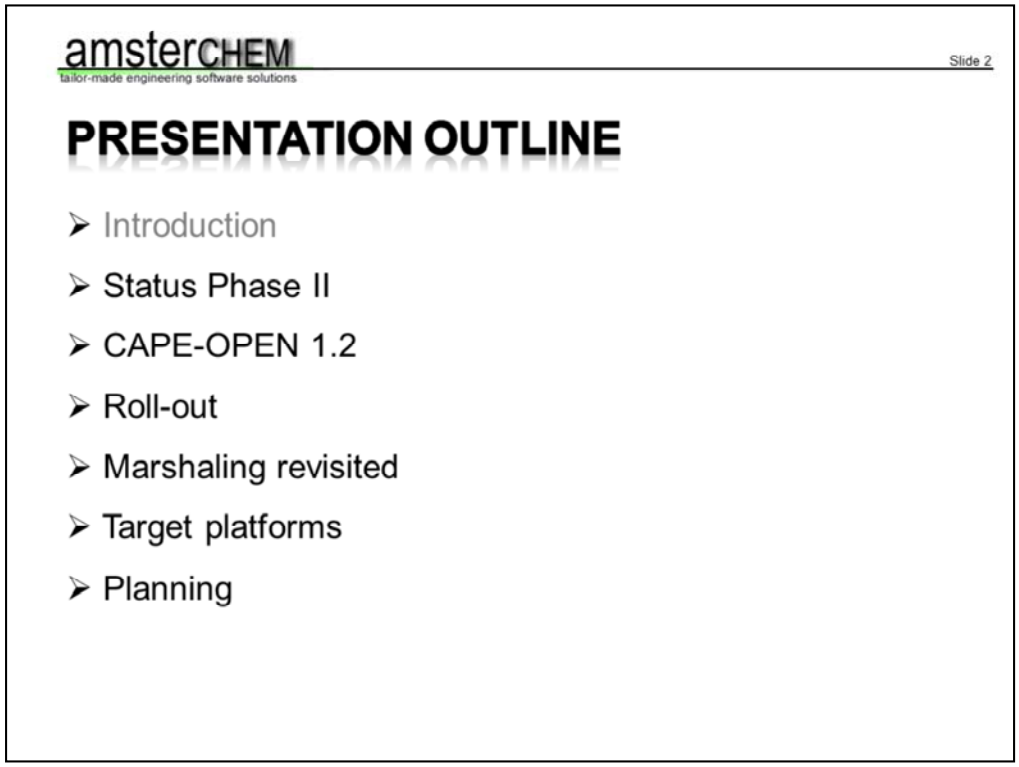
Slide 1

Jasper van Baten – AmsterCHEM
Mark Stijnman – Shell Global Solutions
Bill Barrett – EPA
Michael Hlavinka – BR&E
Michel Pons – CO-LaN

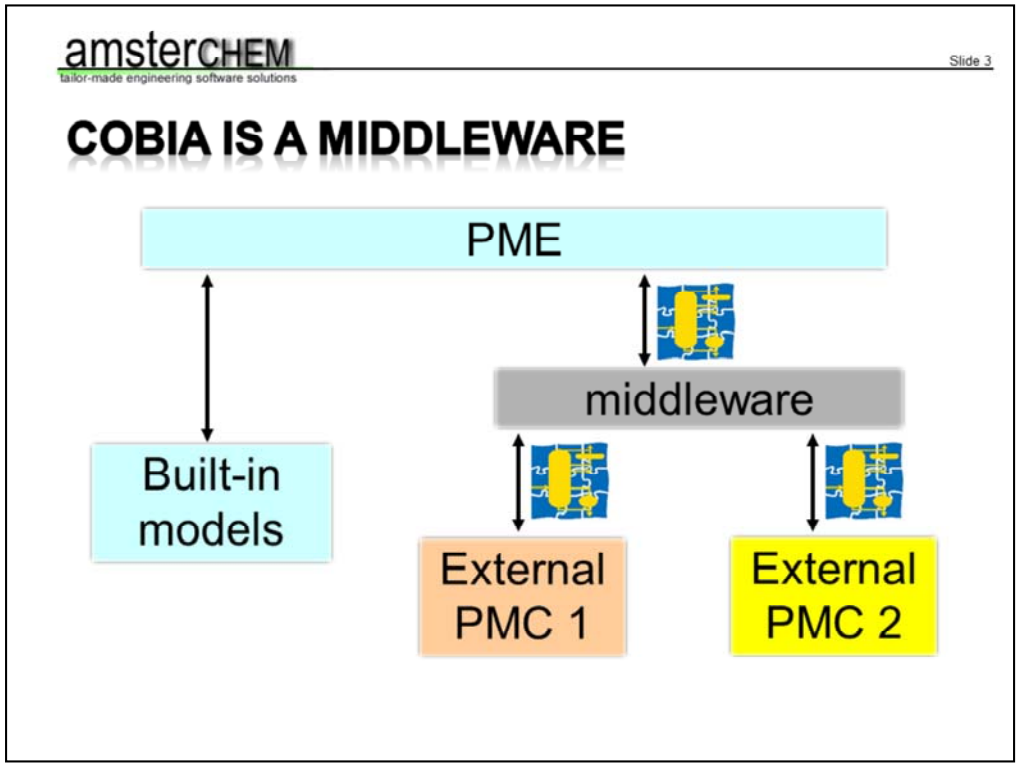
STATUS OF COBIA

CAPE-OPEN Annual Meeting 2020, October 14-15

Good afternoon. My name is Jasper. I am excited this year to provide a status update on COBIA, as a lot of progress has been made. Most specifically, COBIA phase II has finally rolled out, and is ready for use. Let's have a look what that means exactly, and how we will proceed with the next phase.

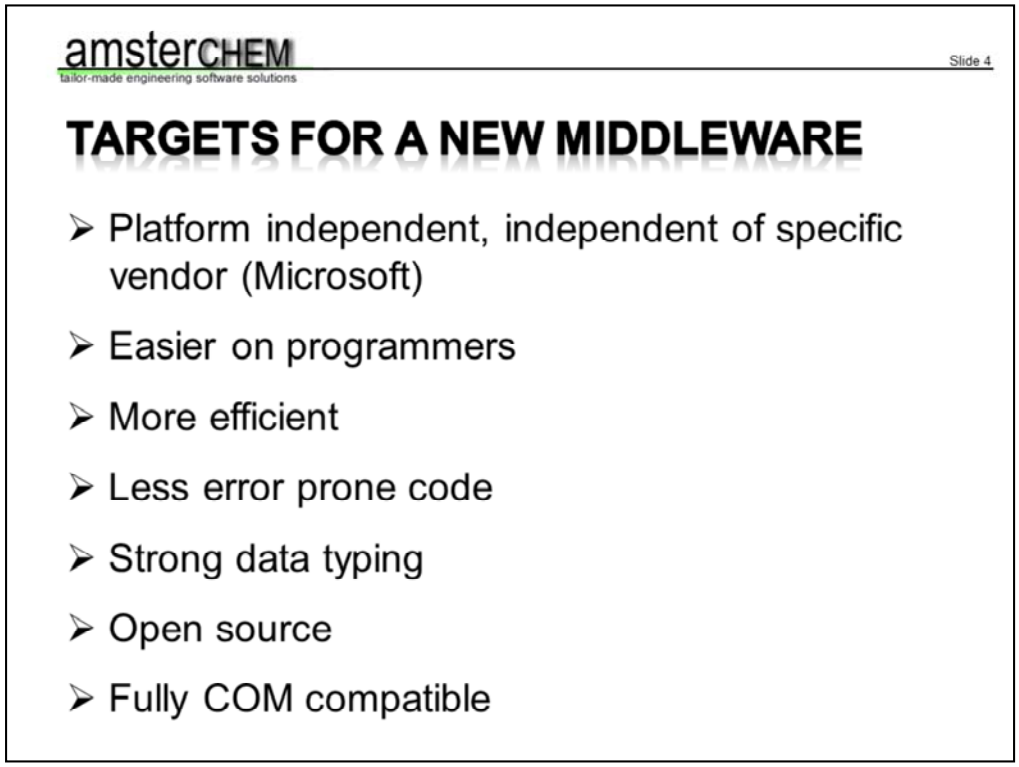


We have a lot of new attendees this year, so I will briefly outline what COBIA is and why we need it. Then I will comment on the status of phase II. COBIA phase II lead to a slight revision of the interface standard, which is of importance to you only in case you base your CAPE-OPEN software on COBIA. Bill already explained the new versioning scheme in the M&T status report, and I will comment here on why we decided we needed a new CAPE-OPEN minor version number and how it applies to new and existing software. Then there is some good news – COBIA phase II has rolled out and is available to everyone now. I will discuss what that means exactly. Then we move on to the future. COBIA phase III will require two main ingredients, marshaling and binding to different platforms and languages. I will finish off by commenting on the planning for COBIA's third and technically final phase.

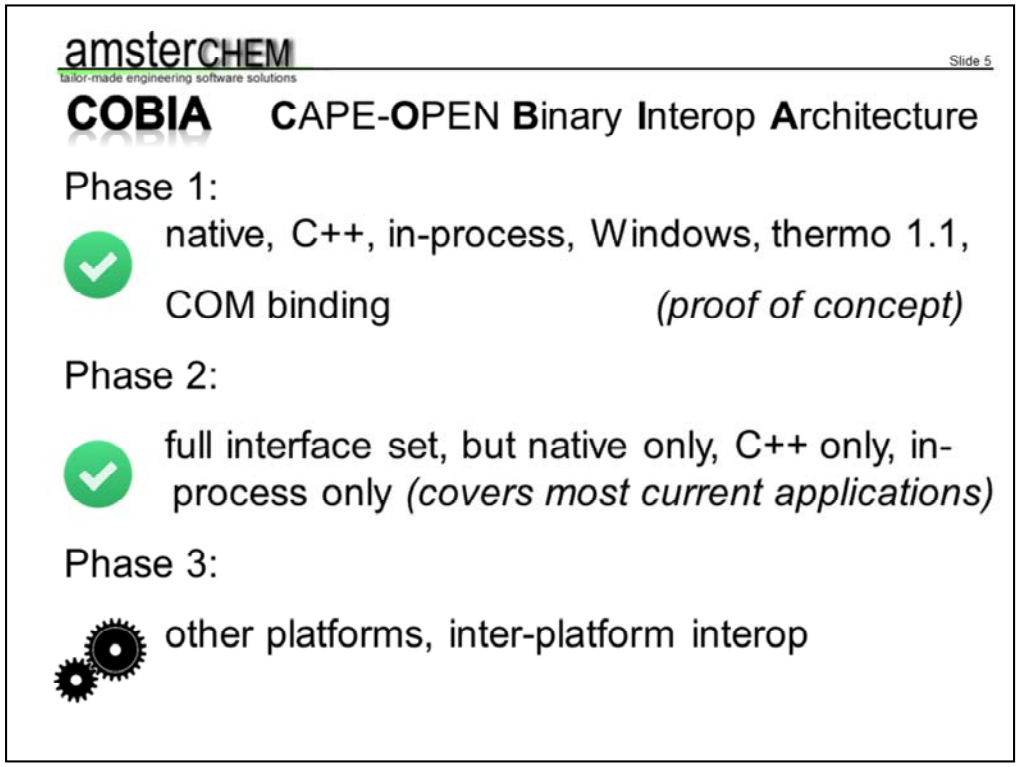


This slide was presented in Berlin, 2011, and Pittsburgh 2012, roughly, and sets the context.

CAPE-OPEN is, in short, a definition of a whole bunch of interfaces that define the functionality exposed by one object and accessed by another object, in terms of which functions are exposed, what are the arguments to these functions and what is expected calling order and behavior of objects that implement CAPE-OPEN interfaces. CAPE-OPEN lives at the boundary between a PME and external modelling components, PMCs. This implies that the PME and PMC can come from different software vendors, that use different compilers, etc. So declaring the interfaces is not enough, we also need to establish binary compatibility between the objects. This is where middleware comes in, it defines calling machinery and calling conventions between two pieces of software. It also provides an API for enumeration and instantiation of PMCs and some other more trivial functionality such as error handling.



These were the design targets for COBIA. We have been using CAPE-OPEN long before the design targets for COBIA were set, which of course means we have already been using middleware in the past. CAPE-OPEN was built around CORBA and COM. By far most implementations were COM based, but COM of course is bound to Microsoft Windows. An open standard should of course not depend on an operating system of a particular vendor. Also we want CAPE-OPEN programming to be easier, more efficient and less error prone. This kind of points in the direction of strong data typing, which was not how CAPE-OPEN's COM binding was originally set up. Because of strong data typing, and shifting responsibility on memory allocation as compared to COM, COBIA programming is easier, and thus less error prone. Finally as CAPE-OPEN is open, we also feel that the middleware should be open. And of course we do not want to ignore the ecosystem of CAPE-OPEN implementations that already exists. So interoperability between COM and COBIA was a design requirement from the start.



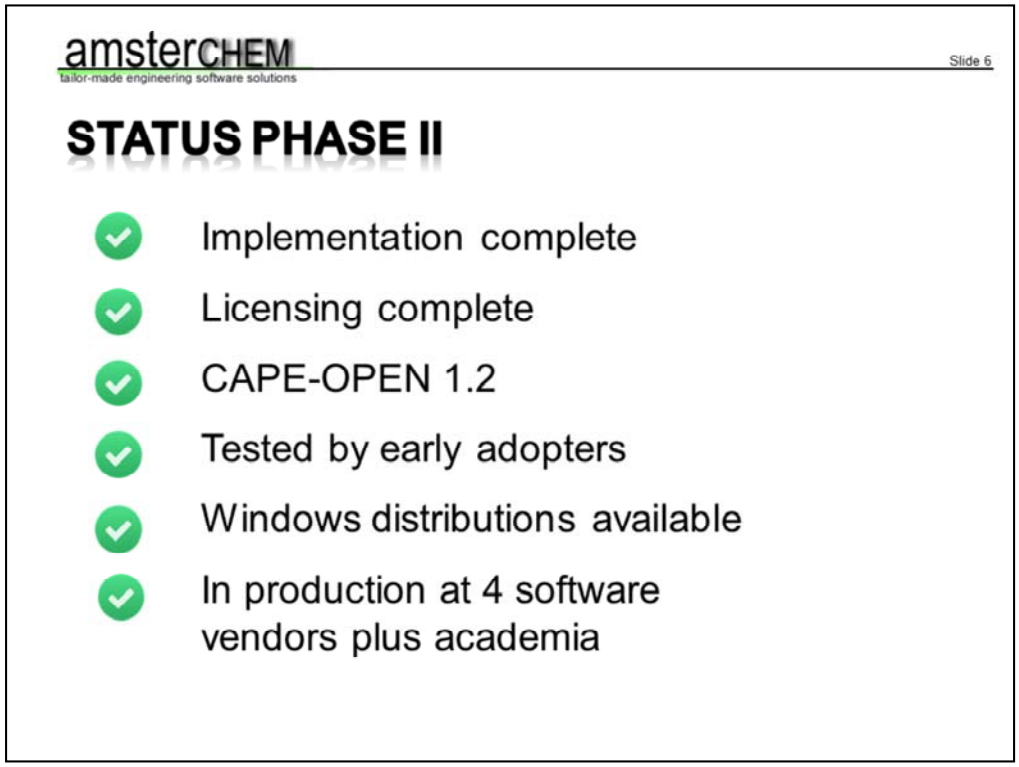
COBIA, which stands for CAPE-OPEN Binary Interop Architecture, was planned in three phases.

Phase I includes prototyping for a subset of the interfaces, on Windows only, thermo 1.1 only, with a test PME and test PMC and full COM interop. It is the proof of concept stage and was completed in 2016.

Phase II extended the CAPE-OPEN interface set beyond just thermodynamics. Still Windows only, and still native only, with C++ as the only language binding. Which covers a good deal of the current CAPE-OPEN application field. This is where we are today.

Phase III will introduce platform independence, additional language bindings, and, as implementations on different platforms need to talk to each other, also marshaling.

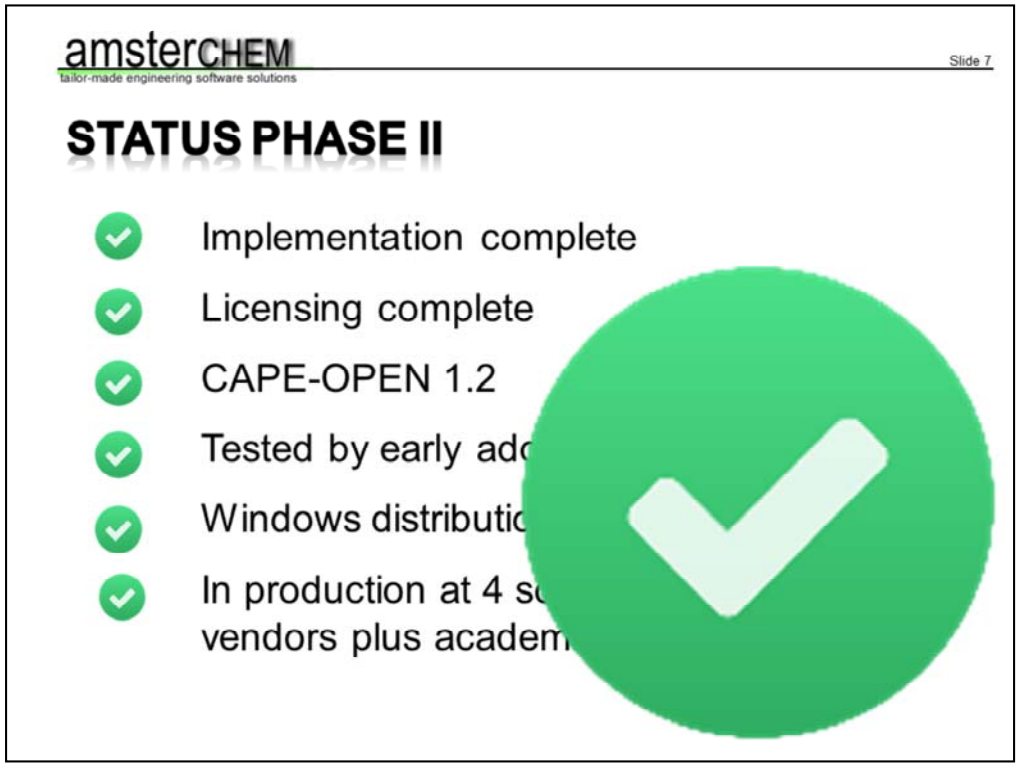
Let us have a look at the status of phase II first.



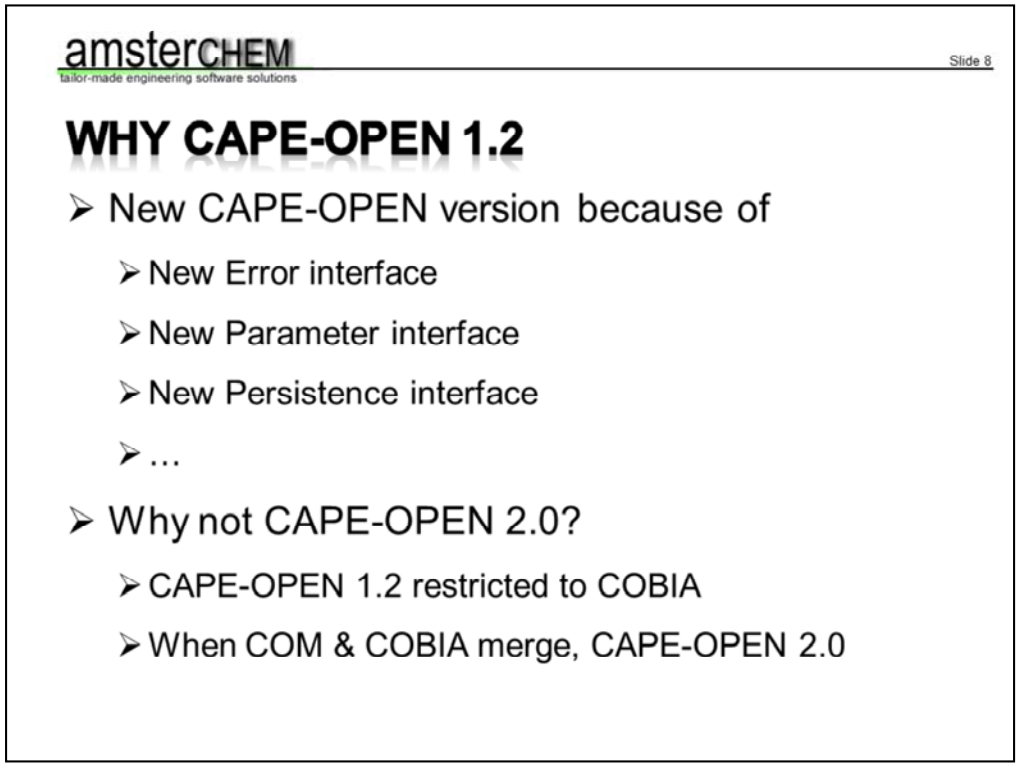
The implementation of COBIA phase II is complete. That is to say, there are new interface proposals we are still working on, that we would like to roll out as enhancements of COBIA based CAPE-OPEN, but where we stand now is sufficient for production software. A big hurdle in getting all of this ready has been licensing. A license format has finally been decided upon earlier this year by the CO-LaN management board. For details you can refer of course to the license conditions that are available with COBIA. In short, COBIA is free to ship out and to use, but some part of COBIA may not be modified, particularly the CO-LaN defined interfaces. The source code of COBIA itself is available to CO-LaN members for the time being.

The COBIA release also brought about the requirement to move to CAPE-OPEN 1.2 – more about that in the next few slides.

COBIA has been tested – initially by early adopters that used a pre-release of phase II back in 2018, partially initiated by a COBIA workshop delivered at the Annual Meeting in 2018. We have come a long way since then, as recently COBIA phase two was made available from the CO-LaN web site, for Windows platforms. COBIA phase two is in active production by two software companies, and about to be taken into production by two more: HTRI and KBC Infochem, both of which will present on their experience with COBIA in presentations later during this meeting.



With all the green check marks, I think we can safely say that COBIA phase II is complete.



So why did we decide it was a good idea to go to CAPE-OPEN version 1.2? When rewriting the COM IDL for existing interfaces to the COBIA IDL, we introduced strong typing, as this was a design requirement. As such a lot of data typing has been made more consistent, and we took the opportunity to correct some other minor things in the existing interface definitions as well.
Some parts were completely overhauled. Particularly: the old error interface of CAPE-OPEN had some limitations and as a result error handling in existing implementations is not always flawless. We decided to integrate error handling much closer into the definition of a CAPE-OPEN object altogether, including an additional function in the very base interface underlying all CAPE-OPEN objects.

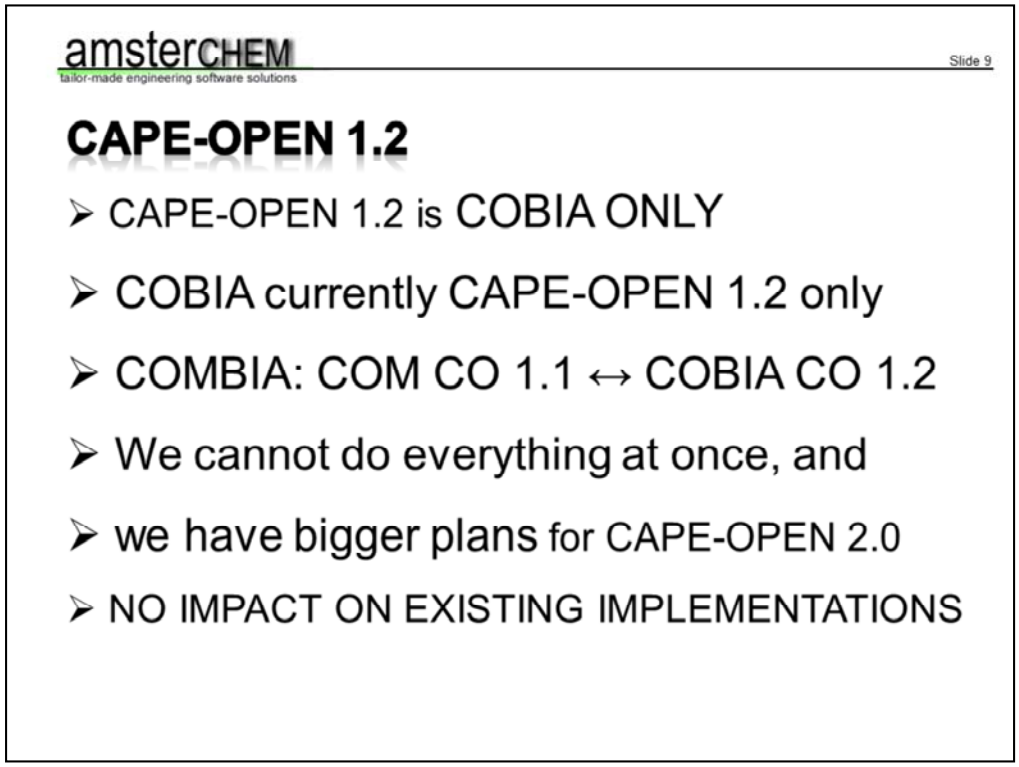
Similarly we had issues with the existing parameter interface, particularly its lack of strong typing. So we rewrote the parameter common interface specification.
Re-using COM based CAPE-OPEN persistence was not even possible, as CAPE-OPEN borrowed COM specific interfaces for that. Now we have a more elegant CAPE-OPEN specific persistence interface.

Some additional interfaces have been replaced, such as the material template system, now replaced by a material manager concept, and the reporting interface which was specific to unit but is now a common interface.

Of course this gives rise to documentation issues if we would have all called this CAPE-OPEN 1.1. Are we referring to the old or new CAPE-OPEN 1.1 parameter interface? A simple solution is to make it CAPE-OPEN 1.2.

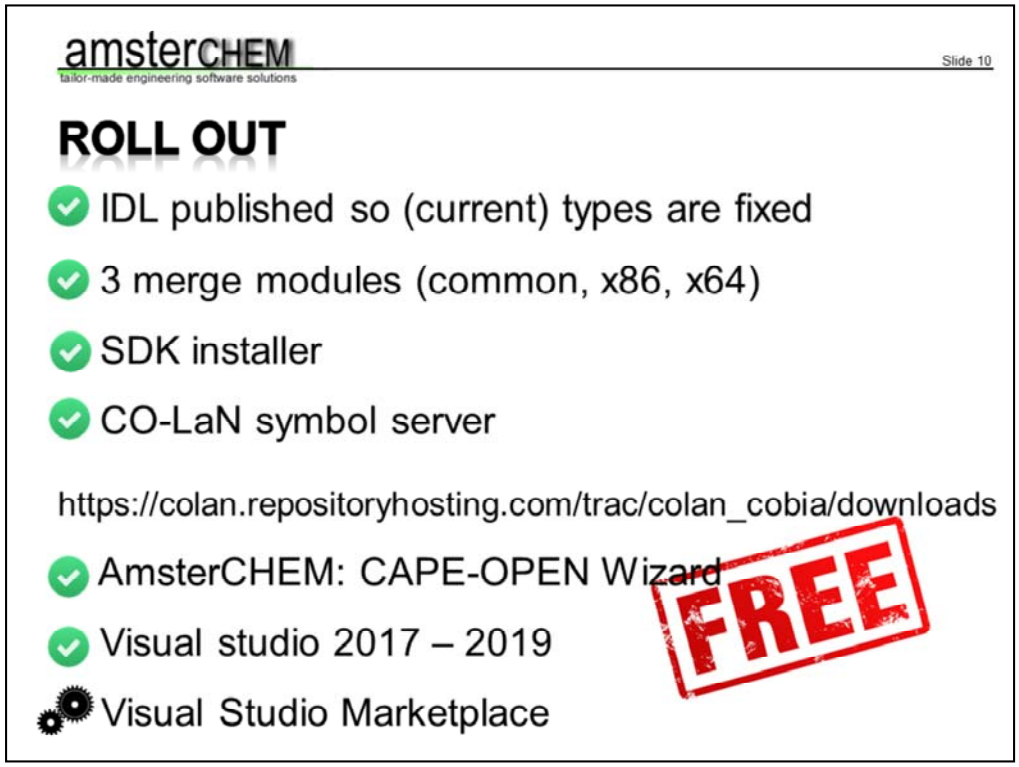
You could of course say, why not make it CAPE-OPEN 2.0? The short answer there is we have bigger plans for CAPE-OPEN 2.0, particularly because CAPE-OPEN 1.2 is currently restricted to COBIA, and for CAPE-OPEN 2.0 we surely want to also support COM.

So what are the implications?



CAPE-OPEN 1.2 has been released in the context of COBIA only. No other CAPE-OPEN interface specification works with COBIA, so COBIA currently requires using CAPE-OPEN version 1.2. Therefore COBIA based CAPE-OPEN 1.2 implementations will work seamlessly with COM based CAPE-OPEN 1.1 implementations. Eventually COM and COBIA will be aligned, but this takes a bit more time for a small organization workforce like CO-LaN to arrange. And then we can also make some other improvements to CAPE-OPEN that are on are wish list.

The important take-away here, is that release of CAPE-OPEN 1.2 has no impact whatsoever on existing implementations. After all, existing implementations are COM based, and cannot even move to CAPE-OPEN 1.2. New COBIA based implementations will automatically work with existing COM based implementations.



So what did CO_LaN roll out?

The CAPE-OPEN 1.2 interface set is defined in the COBIA CAPE-OPEN 1.2 IDL. This means that those types will no longer change, because if we would change these types, new COBIA implementations would break.

COBIA itself is available in Microsoft MSM merge modules, just like its COM equivalent. This is really the only way to release a software components that is shared between multiple applications and manage its installation life time via Windows Installer reference counting.

The software developer kit for COBIA is also released, as a Windows installer MSI. The development installer is only available for x64 systems, which will install support for both win32 and x64 CAPE-OPEN development.

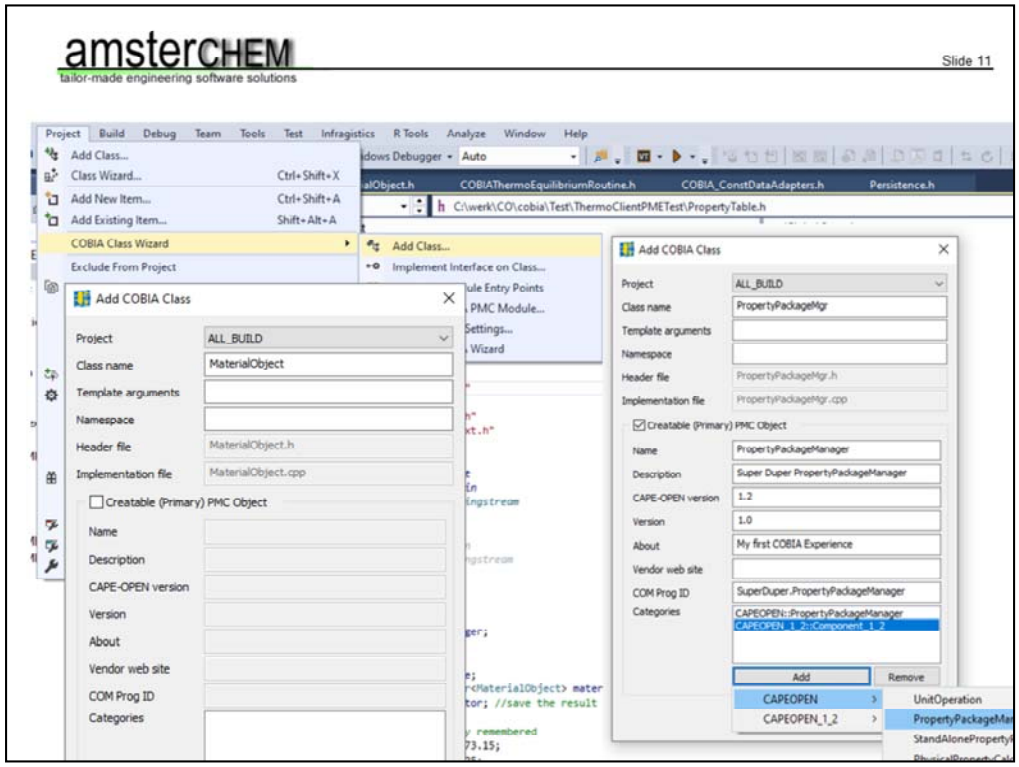
To ensure the ability of debugging software interoperability where COBIA is involved, CO-LaN instantiated a symbol server, at symbols.colan.org. Let Visual Studio know where the COBIA symbol server lives, and COBIA symbols are taken care of automatically.

All of this is available from the COBIA repository. **MICHEL – IS THERE ANY LINK FOM COLAN.ORG??**

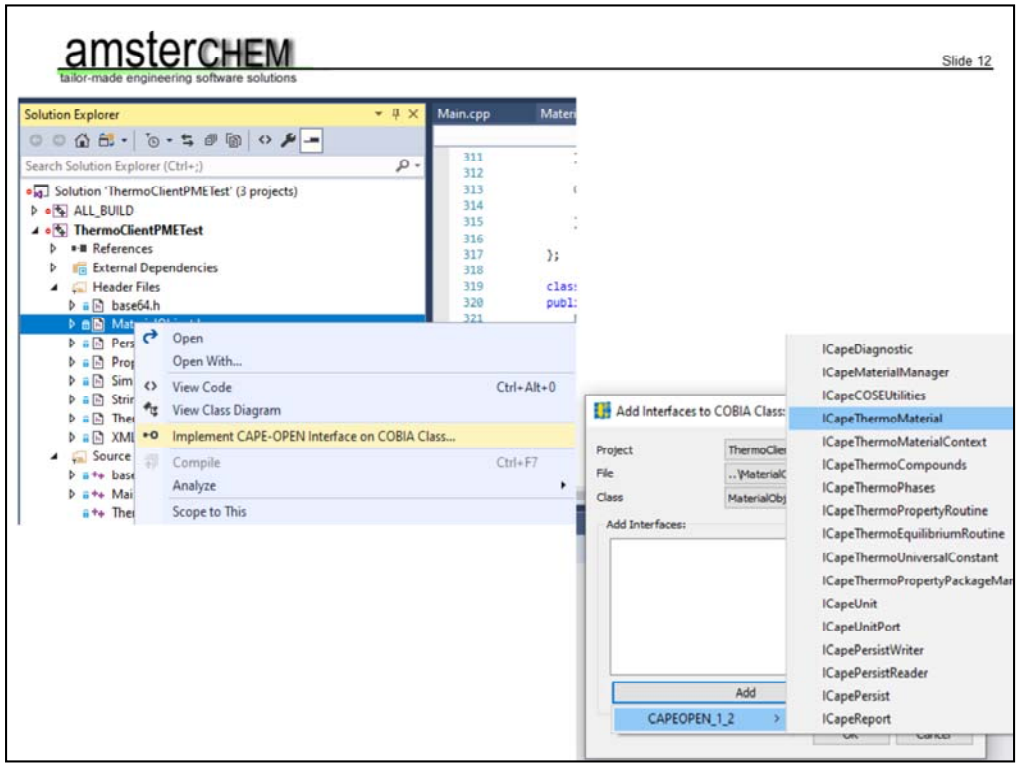
A Visual Studio integration is provided for free by AmsterCHEM. As this is tied in to a commercial product, Microsoft Visual Studio, it is not CO-LaN's place to release such a tool. I plan on making this available from the Visual Studio Marketplace, so that you can directly load this add-in from within Visual Studio, but

I have not yet gotten to it. If you are interested in using it before then, please send me an e-mail and I will provide you with a download link.

So let’s have a quick look at how it all works.



If you have the COBIA SDK and the Visual Studio add-in installed, you can access the COBIA code generator directly from Visual Studio. You can access the COBIA class wizard from various context menus, or the Project menu. From there you can select to add a COBIA class. For an internal class you really only have to specify the name, as shown on the left. For a class that can be instantiated as PMC, one should also provide some registration details, after which the COBIA stub code takes care of your PMC self-registration. PMC functionality is identified via category IDs which can be conveniently added via menus, as shown in the bottom right.

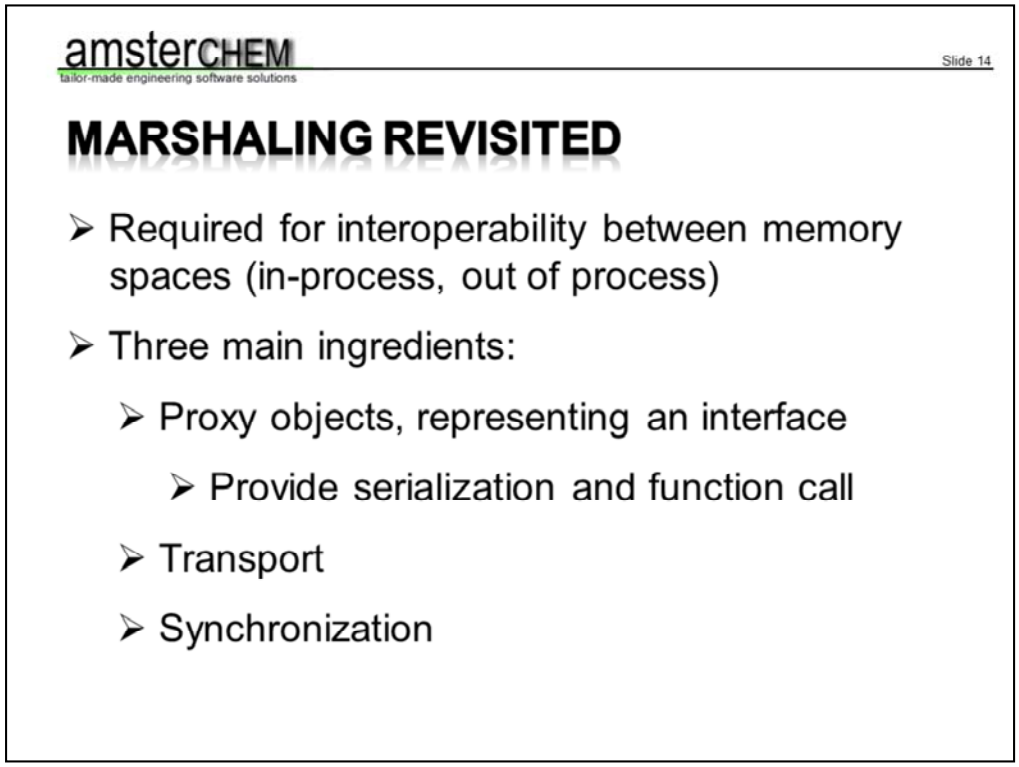


Once a class is there, you can click on it in the solution explorer, and select to implement CAPE-OPEN interfaces on that class. In the dialog that pops up, you can conveniently pick the interfaces from a menu.

Note that the wizard will not tell you which interfaces you must implement for a PMC of a particular type, so you still will need to know a thing or two about CAPE-OPEN, but the programming part is made a lot easier this way.

Whether you go through the wizard or COBIA's command line driven code generation interface, the outcome is the same. Stub code is generated for you, and it is up to you to fill out the functionality. Which, with COBIA's pre-rolled C++ wrappers and adapters is a lot easier than for the COM equivalent, as you can perhaps see for this sample stub code of ICapeIdentification. Wrappers and adapters can also be generated for your custom company-specific interfaces – all you need to do is provide the interface definitions via COBIA IDL.

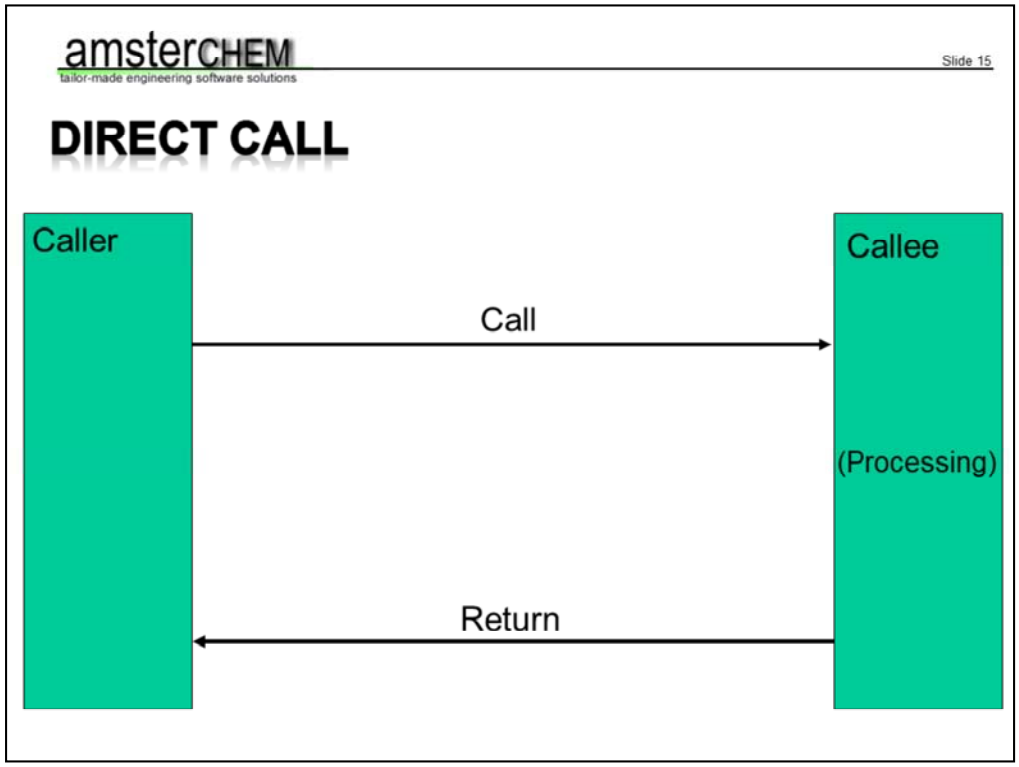
That was a short overview of COBIA phase II – let's move on two the two items that require the most effort in COBIA phase III, marshaling and platform- and language bindings.



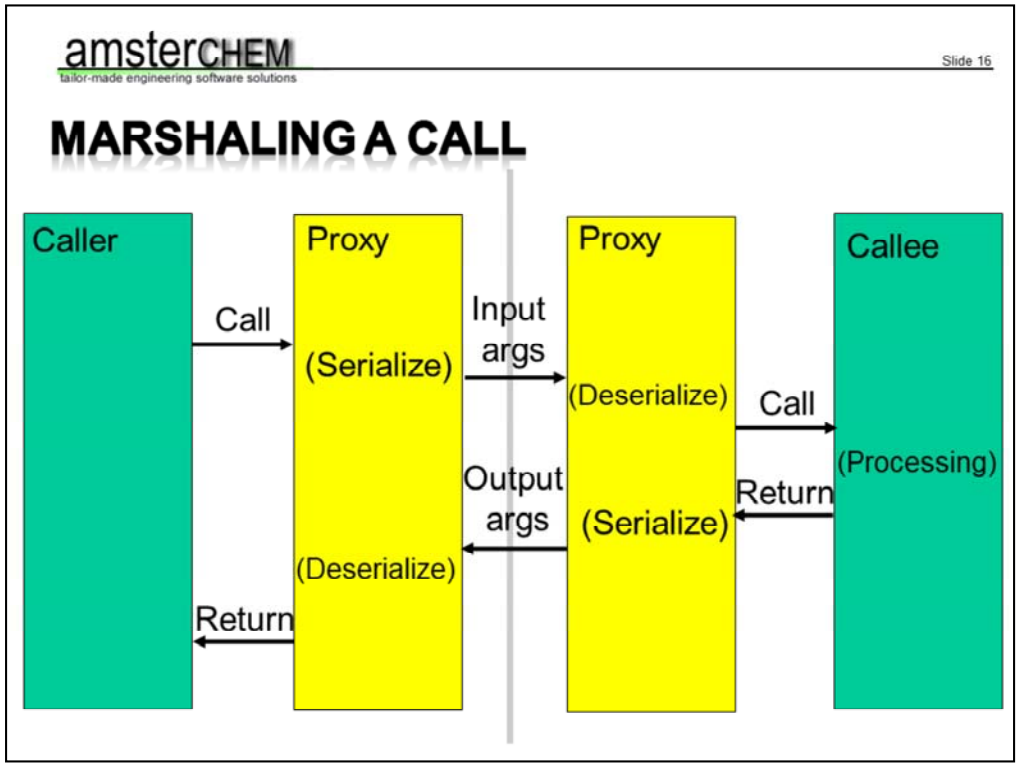
I presented with Mark already our views on marshaling at last year's Annual Meeting. Over time we have learned, and made some adjustments to our vision of how to go about it.

Marshaling we need if two software components need to talk to each other, but they do not live in the same address space. This could be because they do not live in the same process to begin with, or it could be because there are two different platforms in use within the same process, such as a .NET PMC in a native PME.

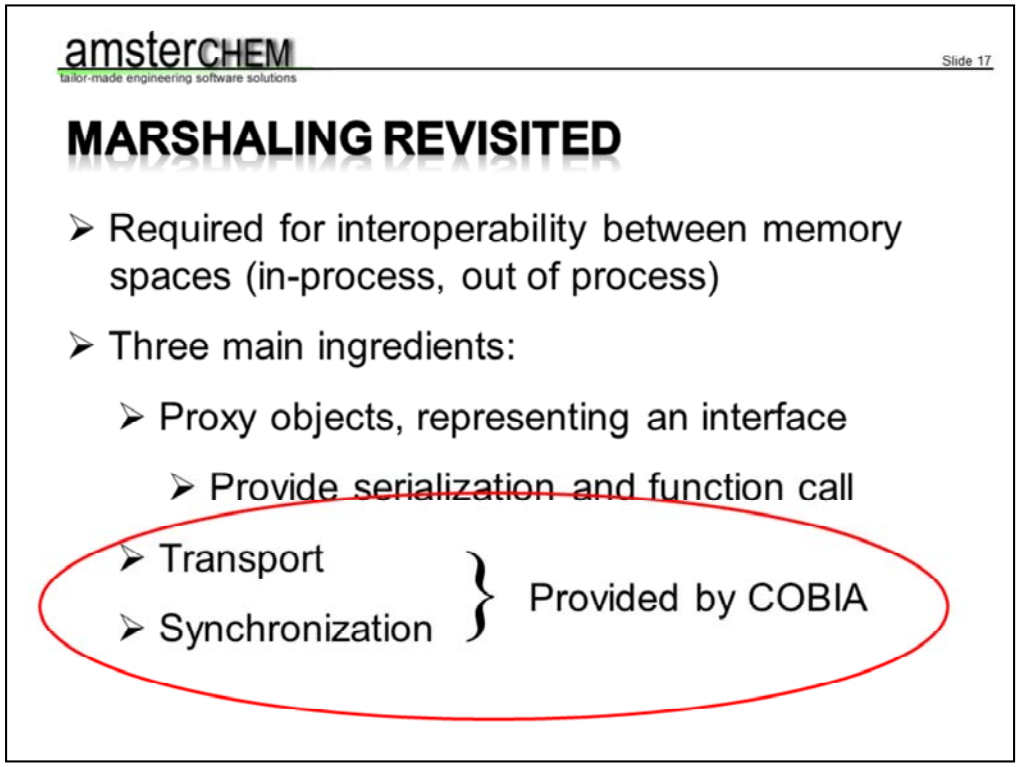
Marshaling roughly requires three ingredients. First we need the dummy objects to actually interface with that represent the actual objects on the other side of the pipe line. We call these proxy objects. Proxy objects take care of serializing function arguments and return values for transport over the pipe line, but are also responsible for making or receiving the actual interface method calls. In addition to proxy objects we need transport, to get the data over the pipe line, and synchronization, to wait until the method call on the other side of the pipe line is finished.



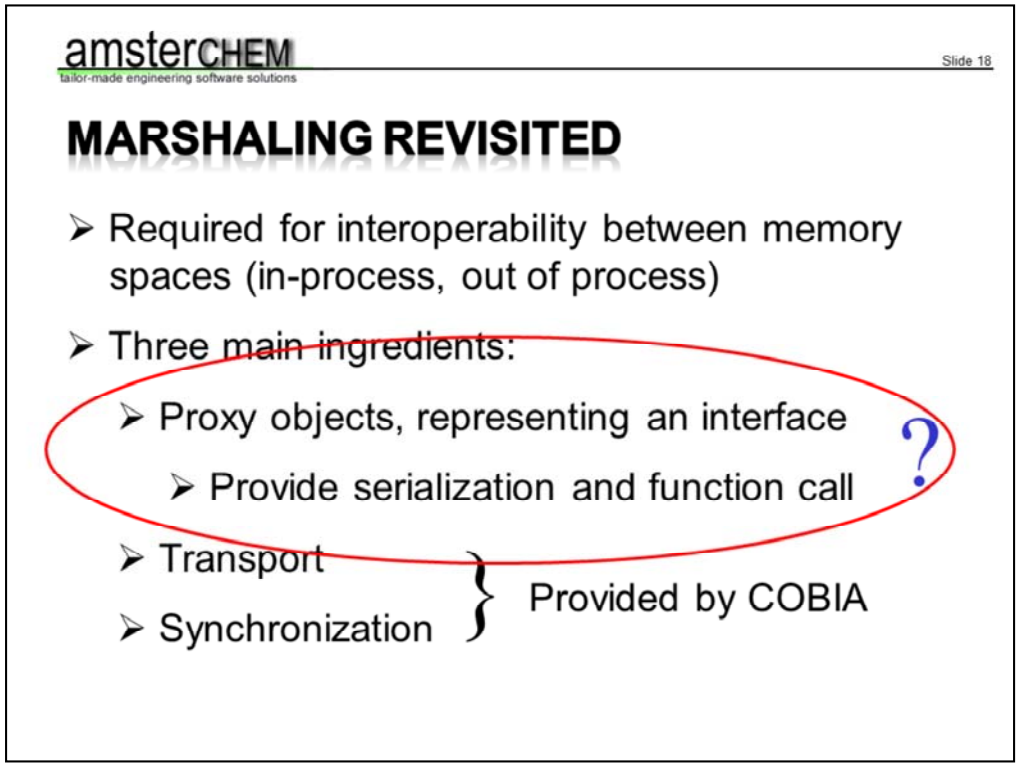
Borrowing two slides from last year's presentation – a direct function call between two objects could look like this. The function call is made by the caller, the callee process the call and returns the function's outputs.



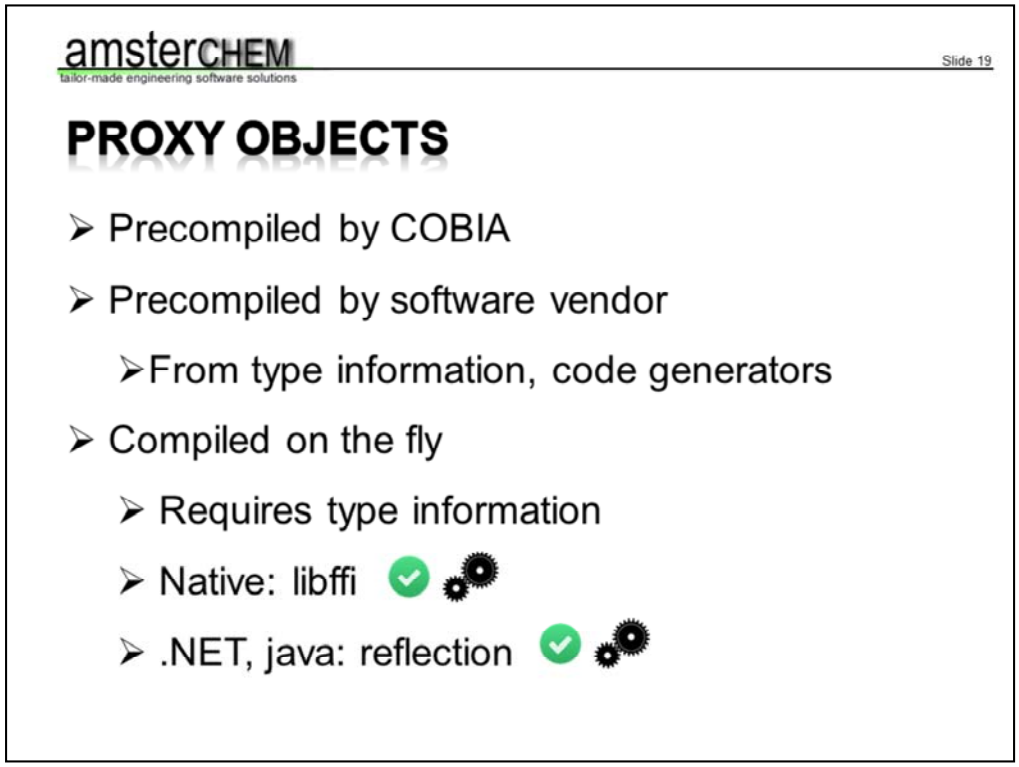
While marshalling, the situation is slightly more complex. Data needs to be transported over the separation between the memory spaces, indicated by the grey line. To get there, the caller calls a proxy object, the yellow block on the left, that represents the actual callee, the green block on the far right. The caller talks to the proxy as if it is the actual object. The proxy must be able to receive the call, serialize the inputs data and send it over the pipeline to the proxy object that represents the caller, the yellow block on the right. This object must be able to actually make the call, after deserializing the data. Then the output values are serialized, sent back over the pipe line to the proxy that represents the callee, which deserializes the outputs and returns it to the caller.



So looking back the main ingredients, we can now conclude that the transport and synchronization are generic, and do not depend on which CAPE-OPEN interfaces are used. These can therefore be provided by COBIA. Of course COBIA will use a different transport and synchronization in-process than it will out-of-process on the same computer, or on a remote computer. So multiple transport and synchronization objects must be implemented, which are then of course best represented by an interface itself, for example ICOBIATransport. COBIA will select an appropriate implementation when the PMC is loaded.



But what about the proxy objects? These callee proxy of course must implement a particular CAPE-OPEN or custom defined interface. And the caller proxy must make calls on that same interface. After the last meeting, the best idea was that all CAPE-OPEN components would bring the proxy implementations needed for all CAPE-OPEN interfaces they exercise, for the platform at which they run. We have adapted that view a bit.



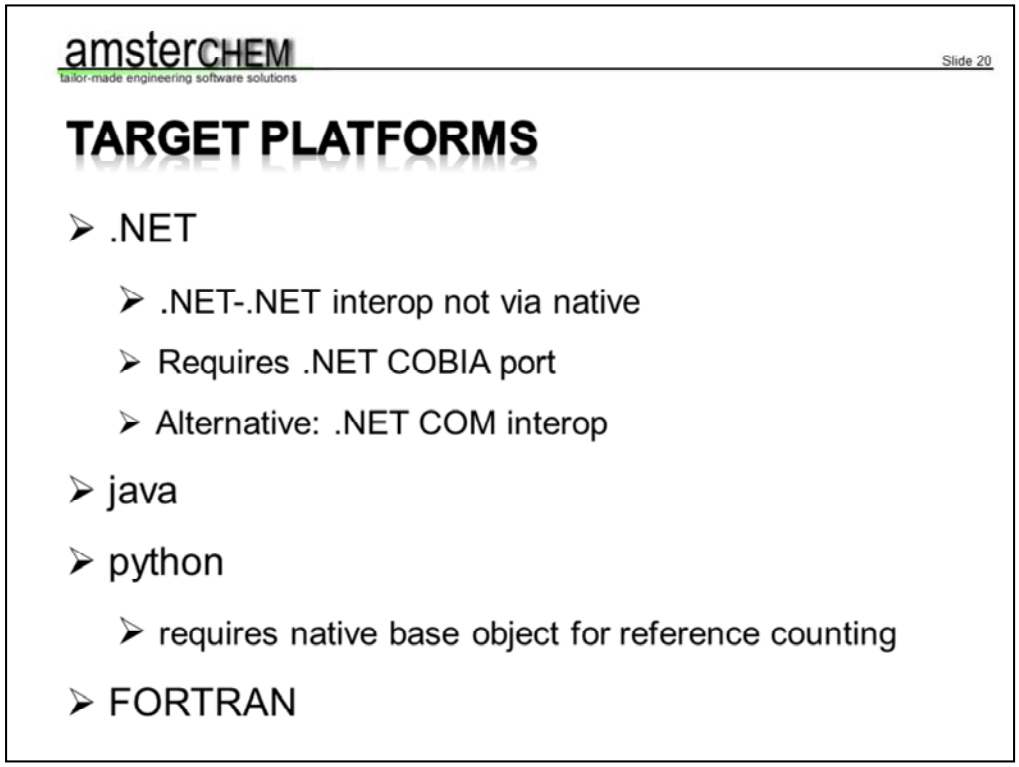
For all CAPE-OPEN interfaces defined in CO-LaN's CAPE-OPEN IDLs, we can simply precompile provide pre-compiled proxy objects as part of COBIA itself. This requires that a class factory for such objects exist inside COBIA that provides the appropriate proxy implementation for the caller or callee of a particular interface.

We cannot do this for interfaces that we do not know of at compilation time of COBIA. But, as these must then be vendor specific custom interfaces, the vendor that uses them can. To create the source code to do so, we can employ COBIA's code generation tools. For a vendor to provide its own proxies could be efficient, but it is also difficult to foresee what platforms such proxies need to be compiled for. So it would be really helpful in case precompiled proxies are optional, and we could get away with proxies that are generated, compiled if you will, at run time. We have looked into this and turns out to be not too difficult, if you have the type information for the interface. Implementing an object or the executable code that calls a function is a rather complex task that is normally done by a compiler. There is a library available that can do it on the fly – it is called libffi, or the foreign function interface library. This is published by Red Hat with a very liberal licence, and I to not think we have to be concerned with future maintenance of this package, as it is used by some players that are bigger than CO-LaN, including Python and OpenJDK. We have provided a proof-of-concept earlier this year for both the callee and caller objects – details are available on request. For .NET and java it is not that difficult at all to generate proxies on the fly – as both of these platforms support reflection. Calling a function with .NET refection is straight forward. Compiling an

object on the fly that can receive a function call with .NET's System.Reflection.Emit – again a proof of concept was put together earlier this year that dynamically generates a caller and callee proxy object just from type information.

This is more or less the plan for Marshaling in Phase III. As you can see – there is a substantial amount of work to be done there.

Let's move on to target platforms.



.NET is on our wish list. We cannot just make all .NET calls go over the native COBIA, because that would imply that all interactions between a .NET PME and .NET PMC would need to cross the native-.NET boundary twice, which would be detrimental to performance. Therefor it would be better to have a .NET version of COBIA itself, and simply translate all CAPE-OPEN interface to .NET as well. Not so difficult at first sight; the .NET version of COBIA can make calls to the native COBIA for all utilities, such as CAPE-OPEN registry access. We could avoid .NET altogether by requiring that all .NET implementations are based on COM rather than COBIA, in which case the usual COM-COBIA interop can be invoked.

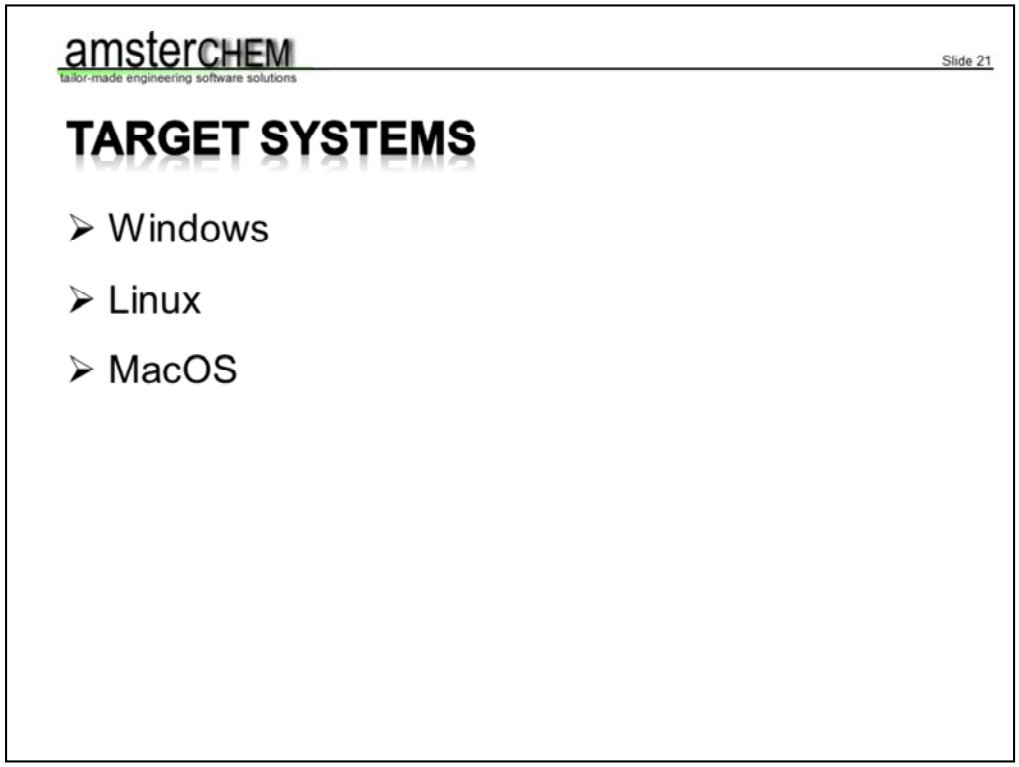
Java is also a nice candidate once we decide to no longer bind to just the Microsoft Windows platform of course. Considerations here are similar, and a java version of COBIA would be helpful to avoid any native based interaction between a java based PME and java based PMC. Here we do not have the back-up plan of using COM.

Python is currently one of the more popular programming language, and differs from .NET and java in the sense that marshaling is not required for interoperability with native components, as Python runs in a native address space. Python based COBIA objects would however need to be derived from a Python class that is written in C++, to allow for proper reference counting of the objects once they are accessed from outside of python.

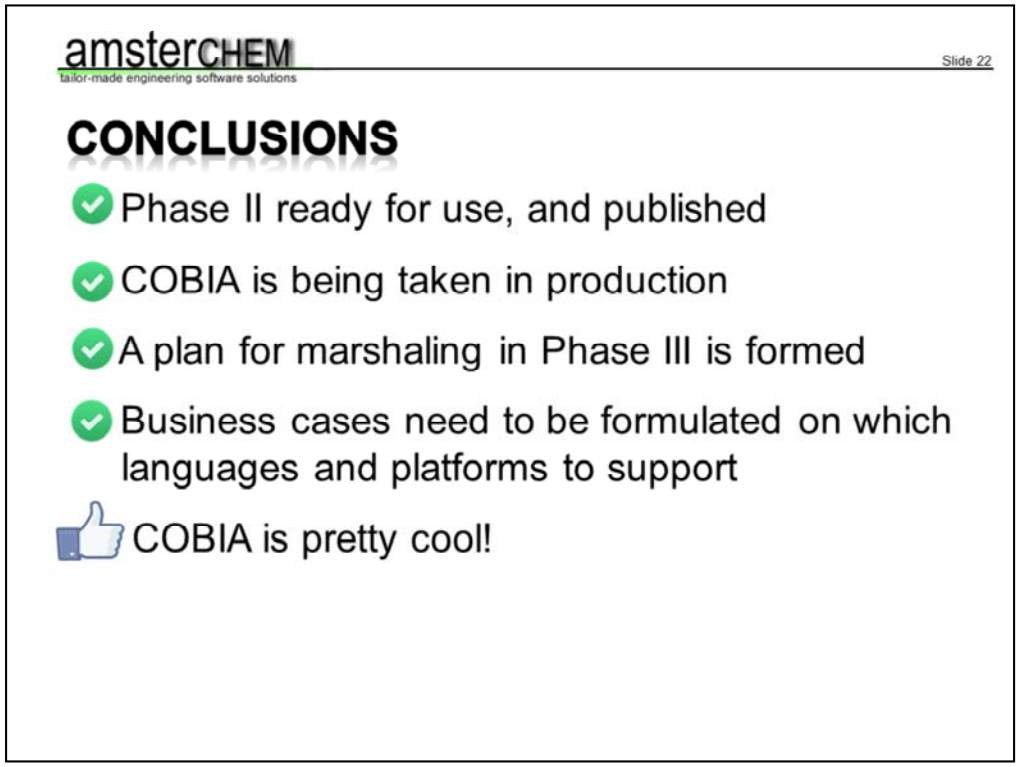
Finally a lot of companies in our field use FORTRAN. As FORTRAN is fully native, this case is a lot easier, and essentially boils down to generating stub code and basic data type implementations that can be compiled by a FORTRAN

compiler.

Whether or not all of these platforms make it into the final proposal for COBIA phase III is not decided yet. A business case will need to be made for all of these. Input from the CAPE-OPEN community is indispensable, so please provide feedback to the M&T sig, the management board or to the Chief Technical Officer, Michel Pons, regarding your company's needs and wishes.

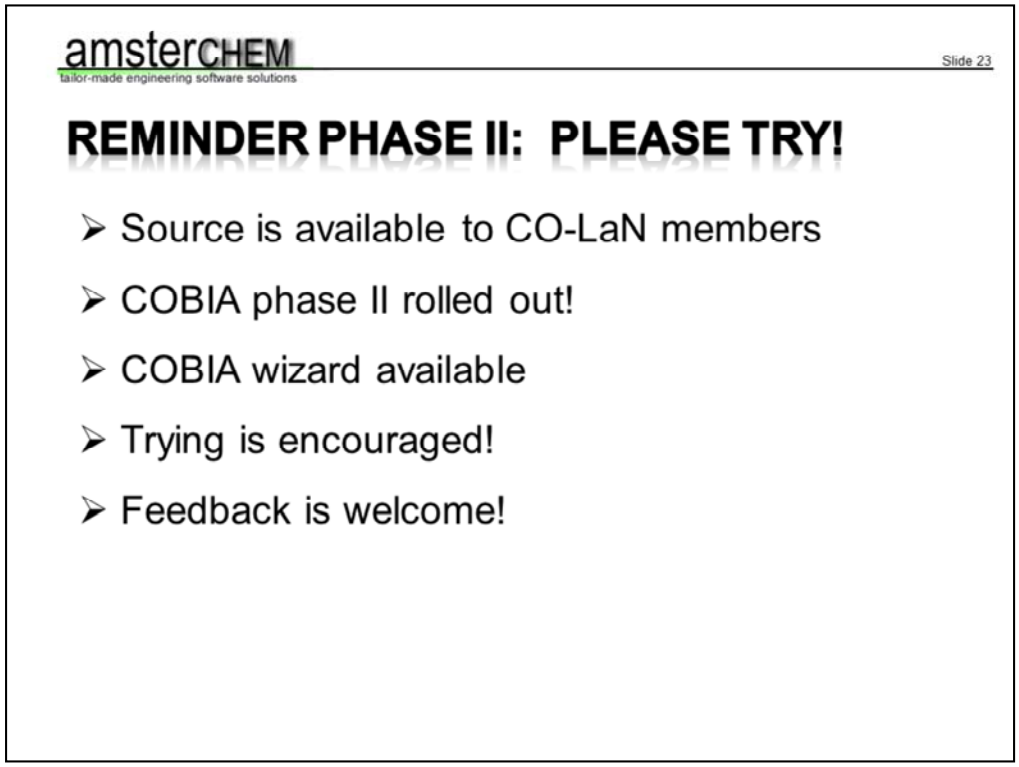


The initial list of target systems seems pretty clear – for the computational marked, most is covered if we support Windows, Linux and MacOS. If you do come up with a business case for running process simulations on e.g. Android devices, please tell us about it.



So that covers all I wanted to share with you today. To summarize, Phase II has been tested, is ready for use, has been rolled out and is being taken into the production stage. We have a plan for marshaling, and initial proofs-of-concept are available upon request. We are still in the process of making a planning of Phase III, which requires making decisions on which platforms and language bindings are to be supported.

Of course also this year I feel the need to convey the obvious – COBIA is pretty cool!



Please try the code that is there. We are working hard to make CAPE-OPEN better, and COBIA is becoming a substantial part of this effort. Based on early experience the benefits of these efforts are exceeding our expectations. But please verify this for yourself, and provide feedback to us where you can.

Thank you for your patience to sit through this – I will be happy to answer questions you may have at this point.