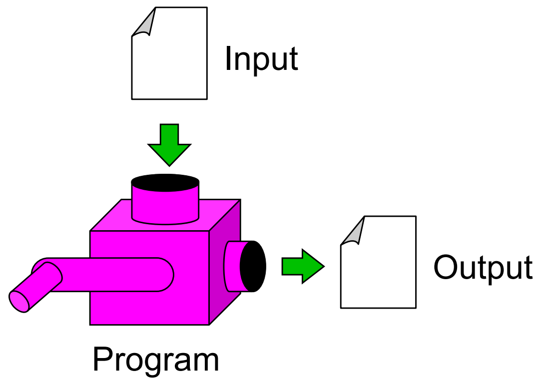
A Pictorial Introduction to Components in Scientific Computing

This is a quick and easy introduction (and justification) to components in the domain of scientific computing.

Listed are the current team members to the components effort here at CASC in Lawrence Livermore National Lab.

My team members call this “The Sausage Grinder Talk”

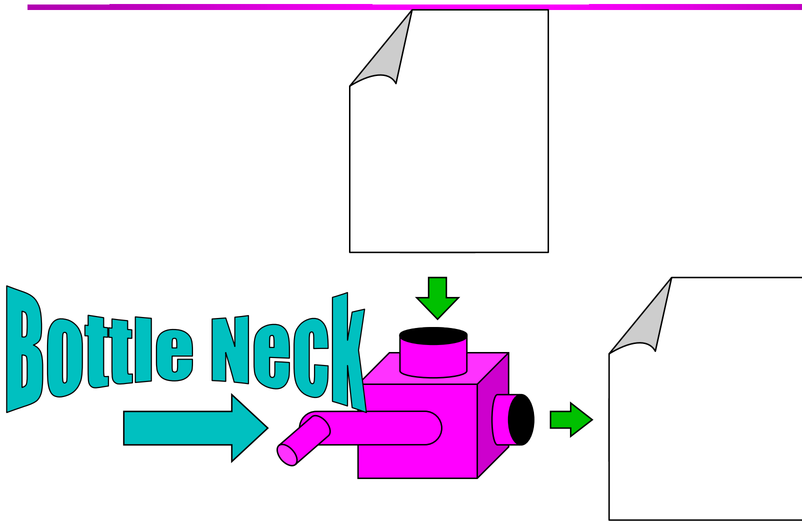
Once upon a time...



Once upon a time, computing was simple

There was a program and you put stuff into it and you get stuff out.

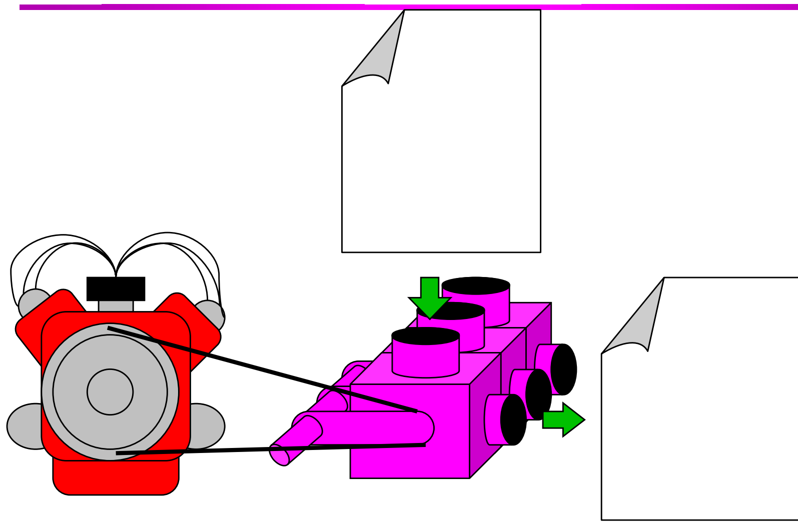
As Scientific Computing grew...



But as scientific computing grew, the data required out of the computations grew.

So then the input grew in turn and the program became the bottle neck.

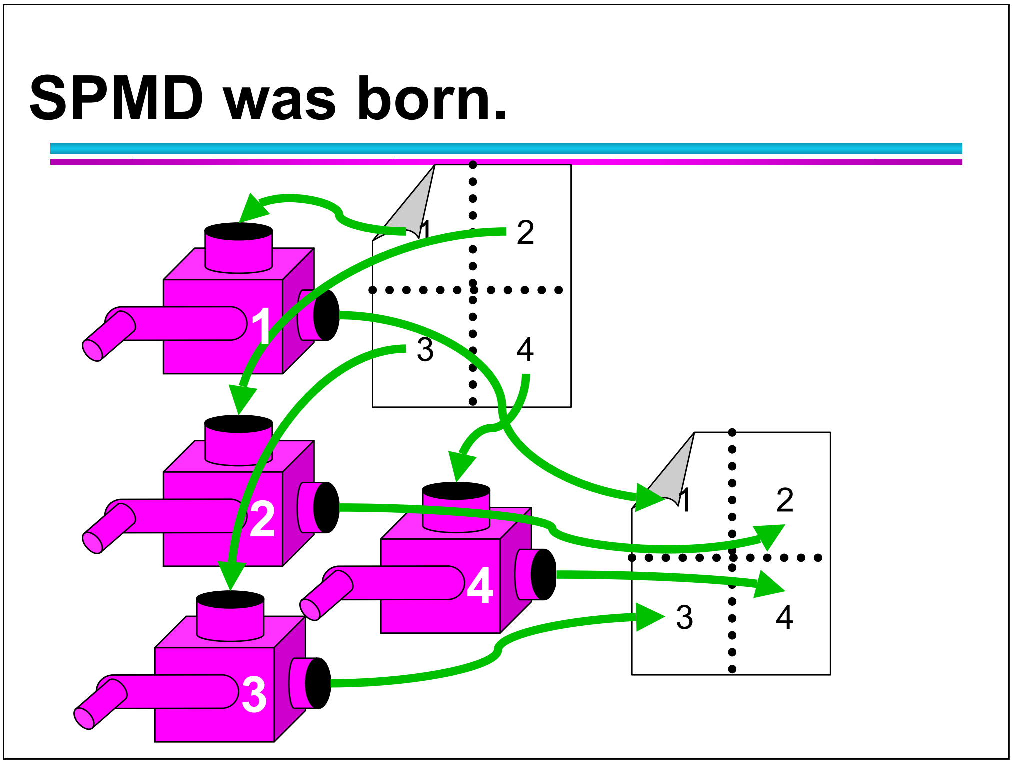
Tried to ease the bottle neck



There were several attempts to ease the congestion.

First was to spend money on custom processors that made the program run really fast. (supercomputers)

Then they started adding more expensive hardware so that one program could do the same thing several times in lock-step (vector supercomputers)

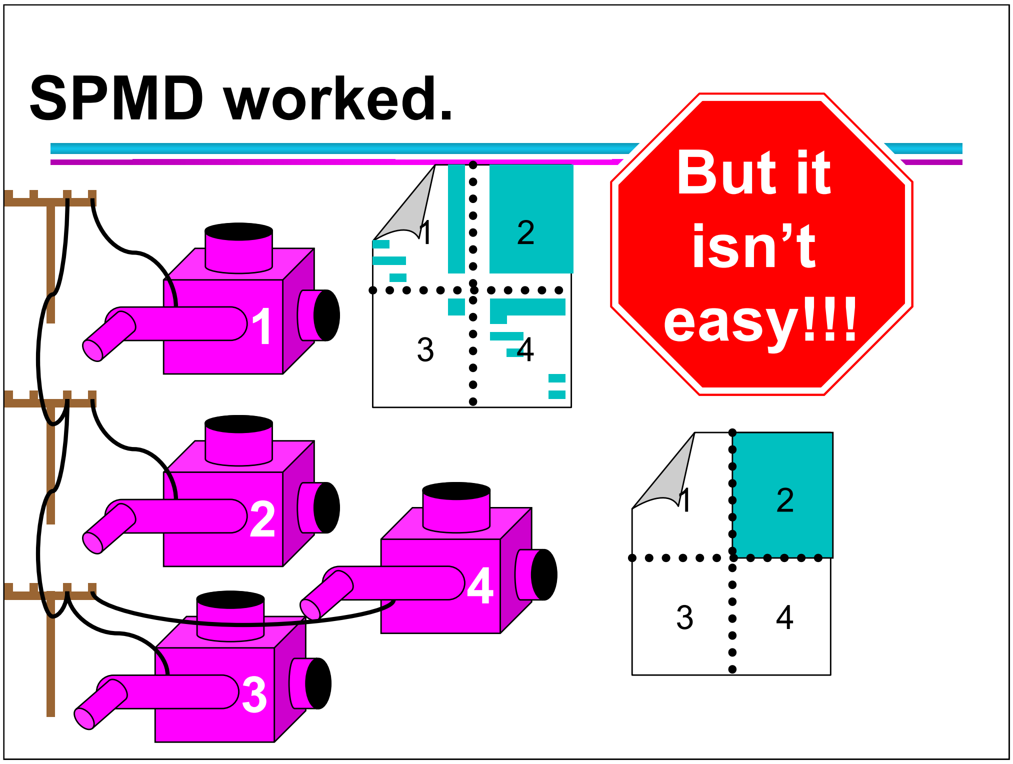


Finally, someone realized how cheap off-the-shelf processors were, and just bought a lot of cheap processors instead of one expensive one.

This meant that the program could run in multiple places at the same time.

This also meant that the inputs needed to be cut up and distributed among the different processors and the output had to be reconstructed from the resulting fragments.

Thus, SPMD was born, Single Program Multiple Data.



But...

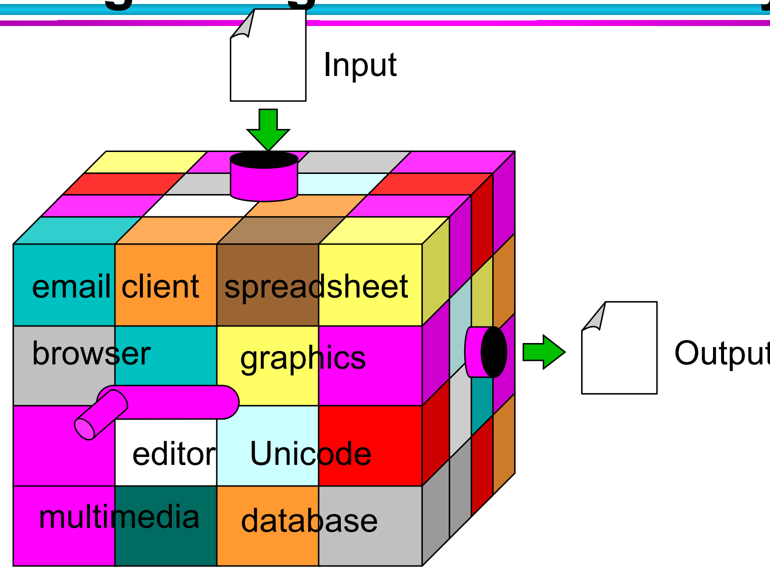
As everyone knows there are very few problems like SETI@home where each part can be computed independently.

For our problems of interest, the data for a single output piece (number 2) has dependencies interwoven throughout the data.

So we set up message passing, which basically means that each instance of the program finds data that it needs and data that it knows to transmit and they call each other and exchange information.

This is “state of art” today in Scientific Computing. SPMD on Distributed Memory, message passing systems.

Meanwhile, corporate computing was growing in a different way

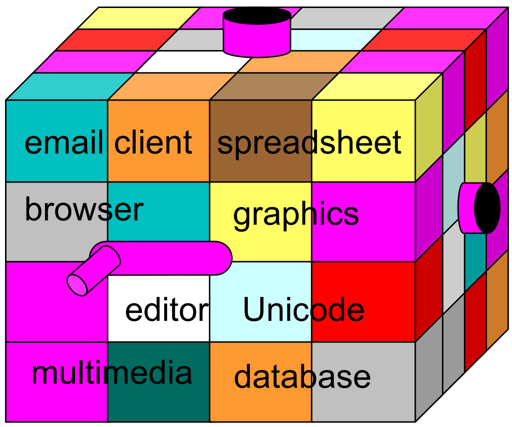


Now, separately from Scientific Computing, Business computing really took off in the last couple decades.

However, it grew in an orthogonal direction.

The input and outputs for programs (say a word processor) didn’t grow by a dozen orders of magnitude, but the application used to construct these documents did!

This created a whole new set of problems → complexity



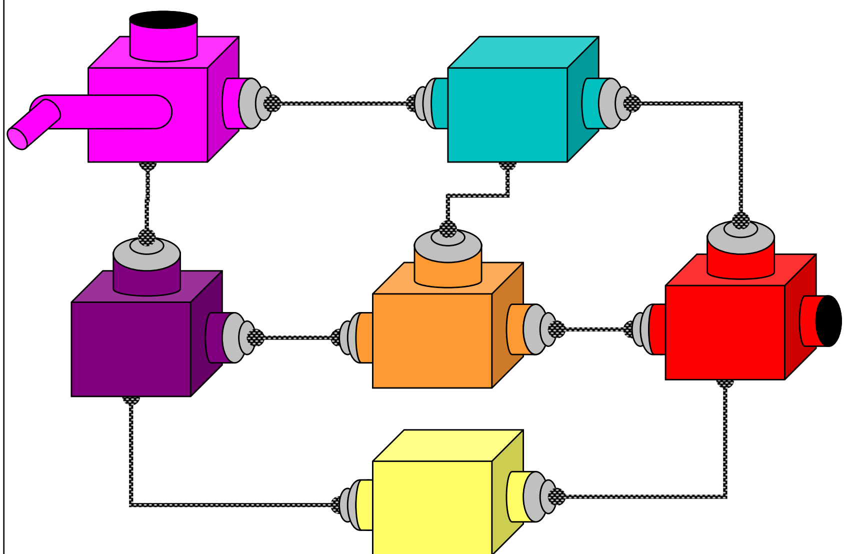
- Interoperability across multiple languages
- Interoperability across multiple platforms
- Incremental evolution of large legacy systems (esp. w/ multiple 3rd party software)

This created a whole new set of problems.

These are the three that I want to concentrate on today. You will see them again in this talk.

They are ....

Component Technology addresses these problems



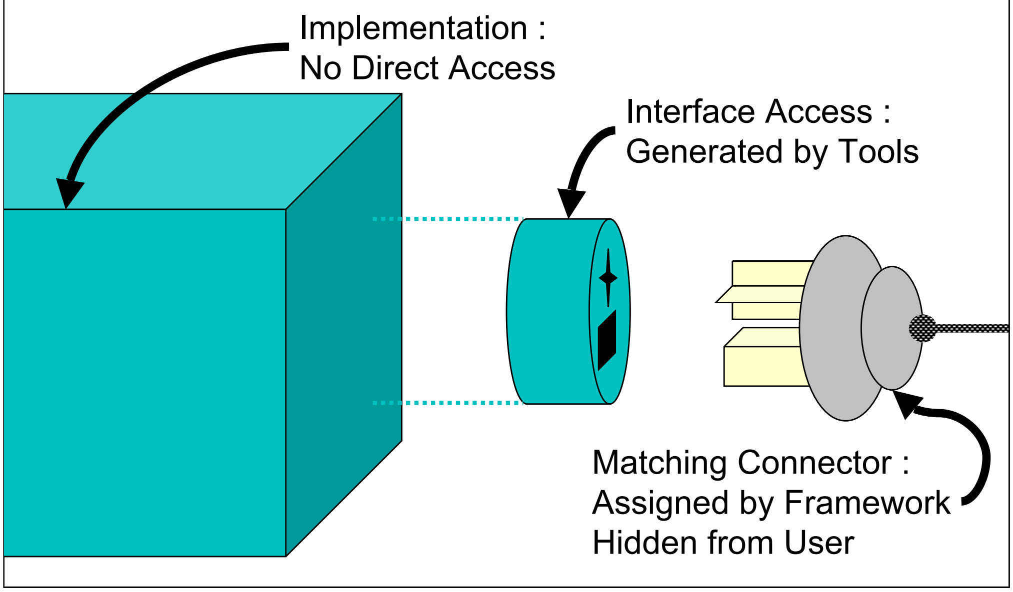
So this is how I draw component software.

The key word to remember about components:

Loose coupling

Let me explain what this drawing means

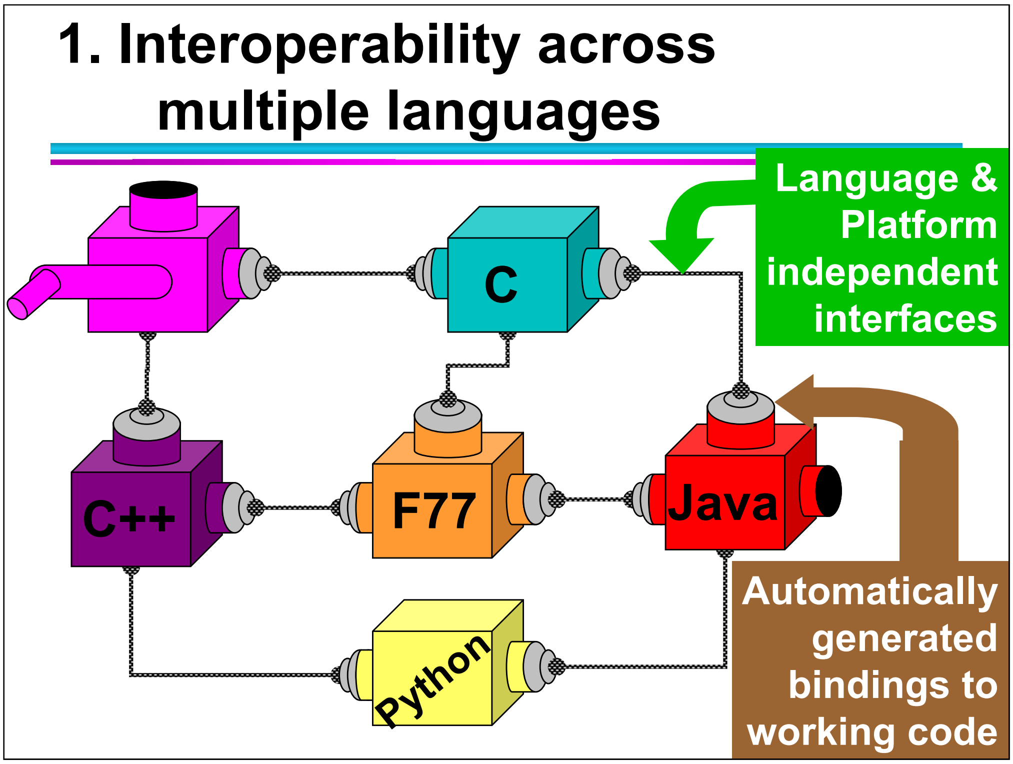
So what's a component ???



The box is the developer's software. It remains (essentially) unchanged.

This socket that is attached to it is an interrface. It is usually generated by some tools.

The connector on the right is assigned to the component by the framework, possibly at runtime.

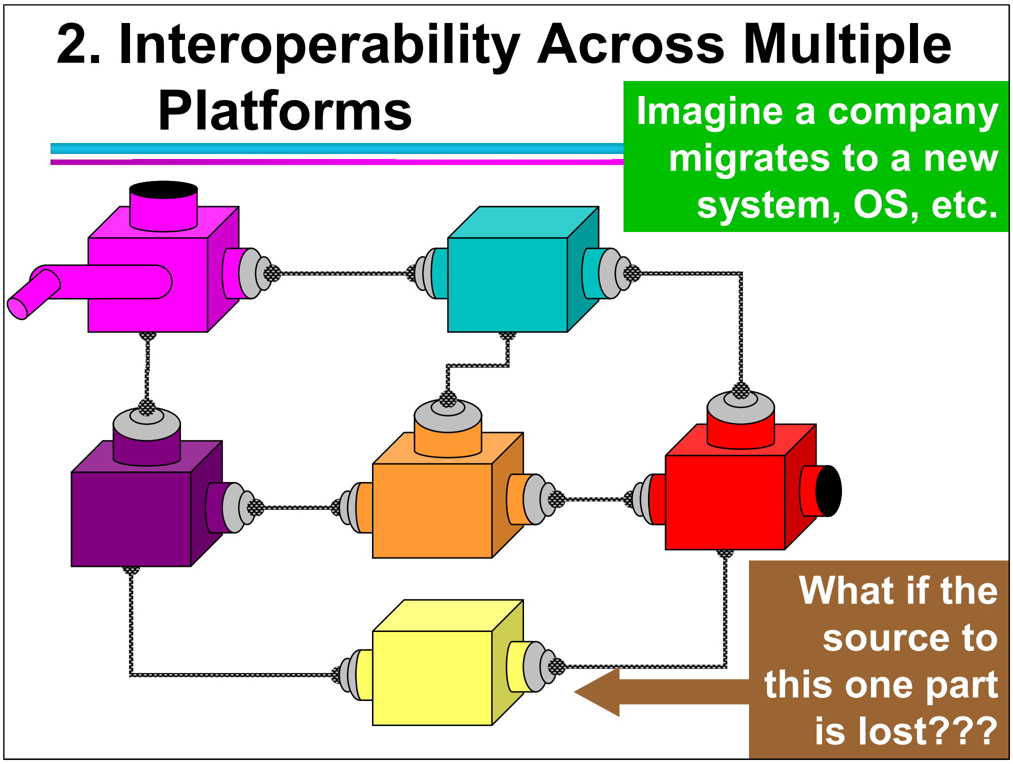


I told you that we'd discuss three problems in detail. Here's the first.

So let me add the languages

And that's it. There's no problem. Each box may be another language, but that's an internal detail of the component.

The "wires" are language and platform independent, and the generated interfaces to the actual code do all the translation between the wires and the particular implementation language.

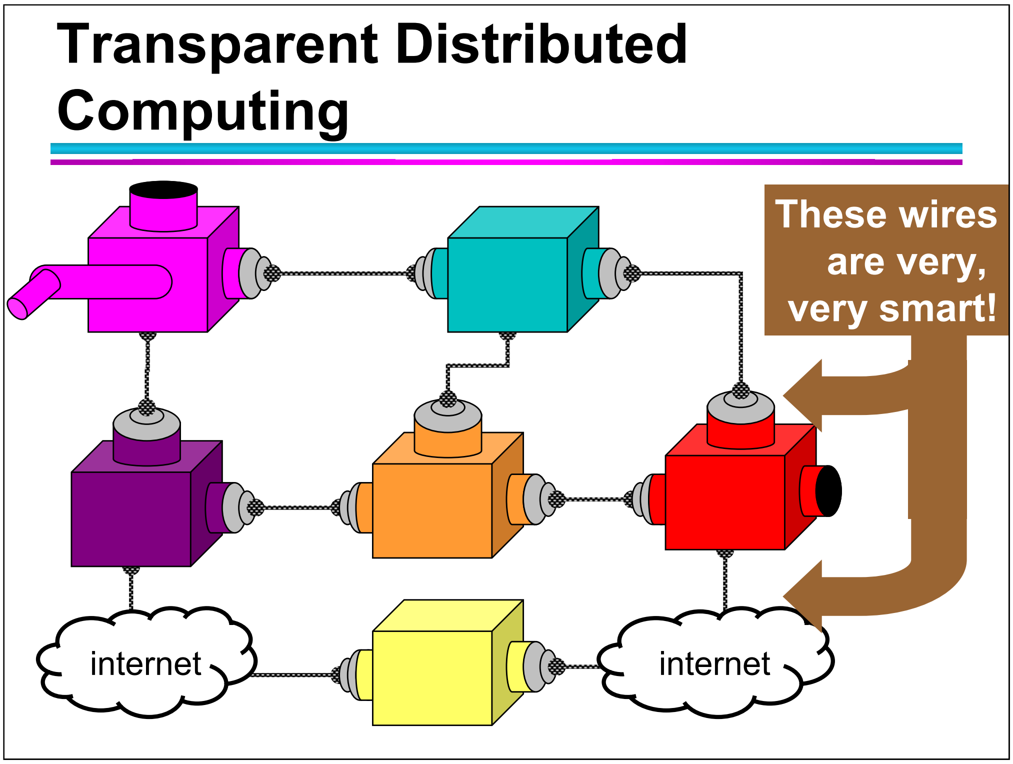


Now a slightly harder problem.

Imagine....

What if...

(I’ve seen this happen to companies)



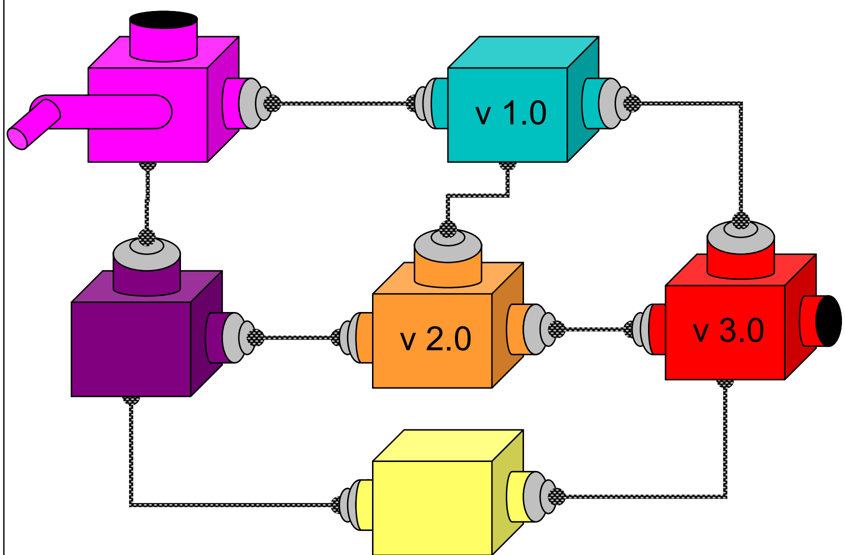
In components, the yellow block would still have to run on the old platform, but the others can move over and communicate over the internet.

More importantly, this can occur without any changes inside any of the boxes, a.k.a the implementation.

Just as the interfaces hide the implementation language from the wires... they can hide the type of wires from implementation.

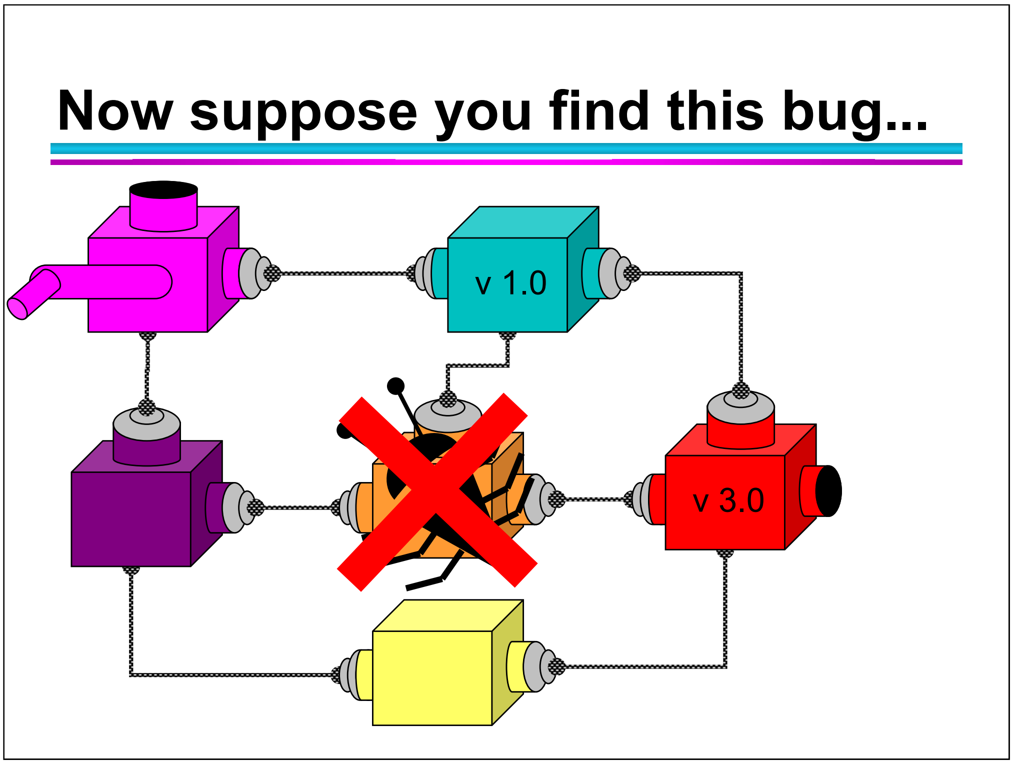
These wires can be complicated things and not just simple communication paths.

3. Incremental Evolution With Multiple 3rd party software



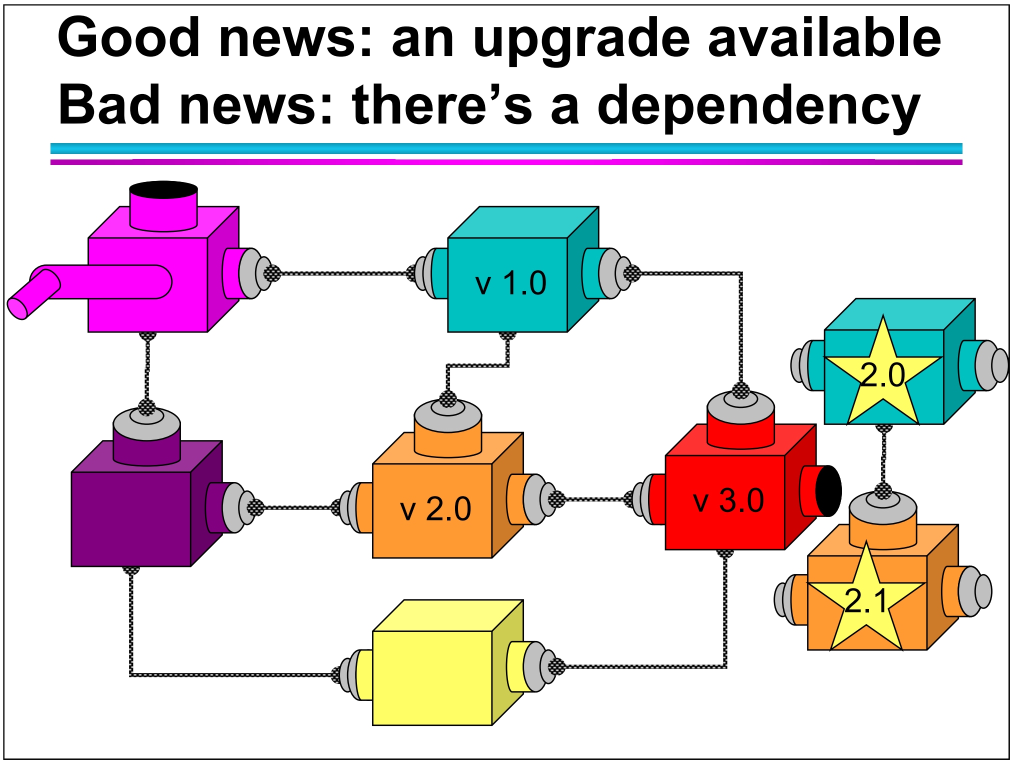
Okay, one last problem with incremental evolution and how industry components handles this.

Let’s start by putting some version numbers on these components



Now imagine you find a bug on the orange component right there in the middle.

Now your whole system is broken, because you need this fixed.



The good news is that there is an upgrade available

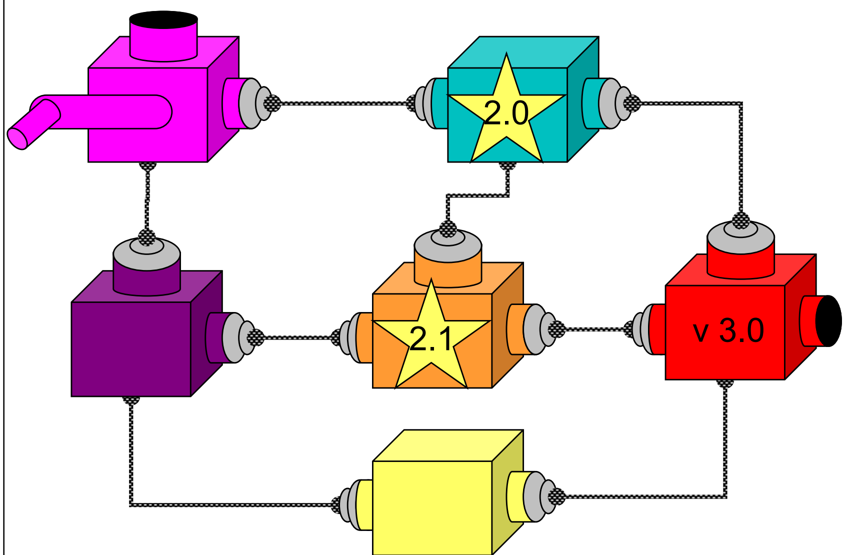
The bad news is that it depends on a new (and incompatible) version of the teal component at the top.

Now you also have the red component which hasn't yet upgraded to teal 2.0.

How many have run into these kinds of situations?

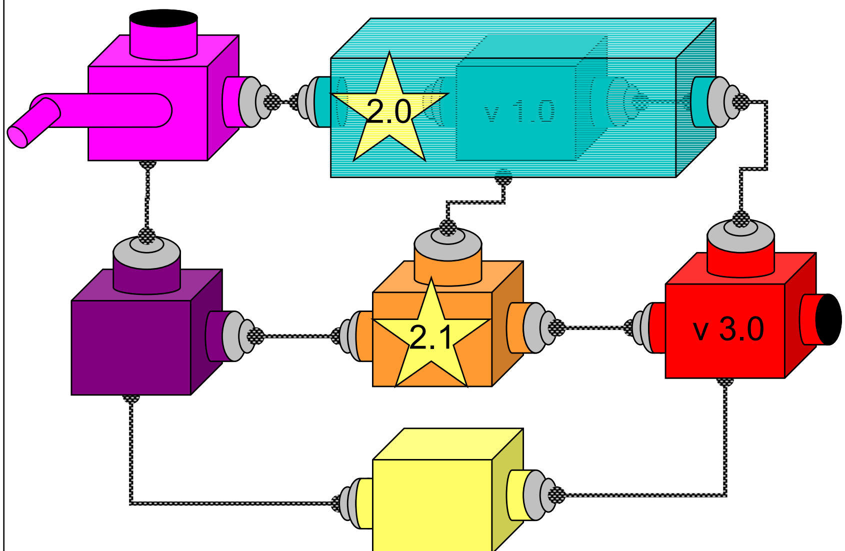
What can you do?

Great News:
Solvable with Components



With components, this will still work.

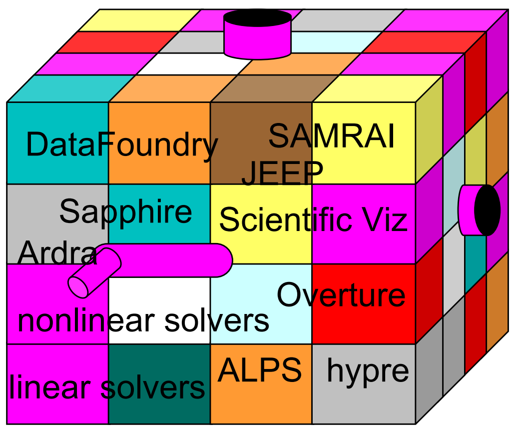
Great News:
Solvable with Components



The trick is (at least with COM) that new interfaces still hold references to the older interfaces under the hood.

The component framework can detect the version mismatch and have the component drill down to its older interface underneath.

Why Components for Scientific Computing → Complexity



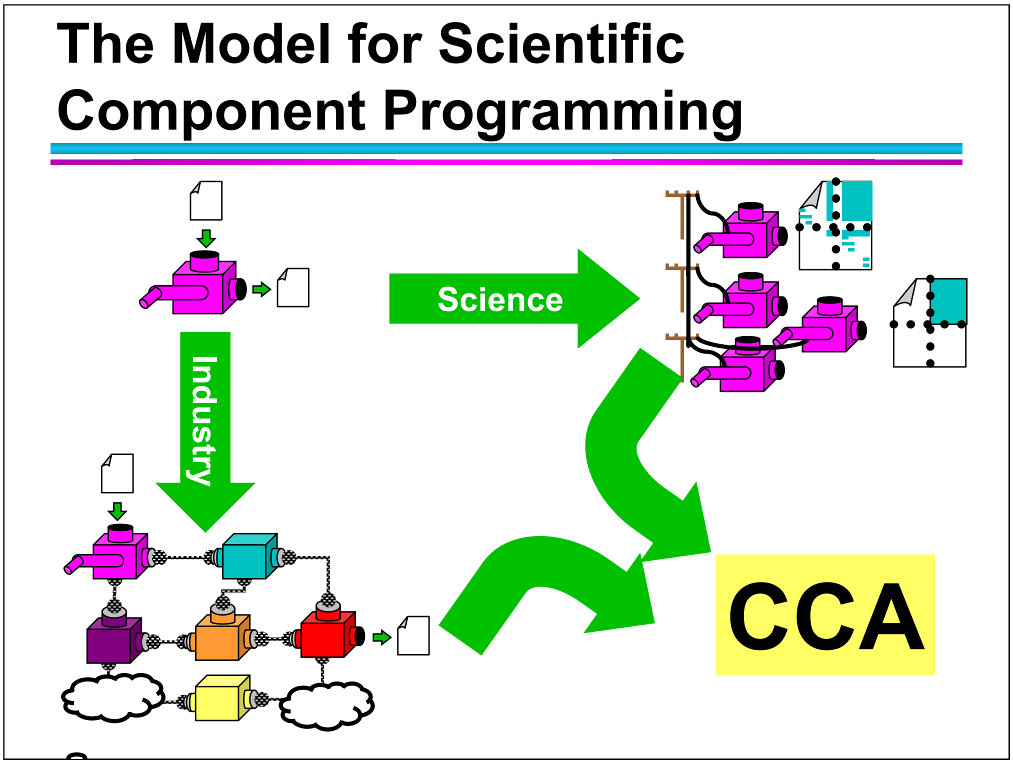
- Interoperability across multiple languages
- Interoperability across multiple platforms
- Incremental evolution of large legacy systems (esp. w/ multiple 3rd party software)

So now let’s take a step back and ask...

Components seems effective in industry, but what has that to do with scientific computing?

Well, if you look the way we’ve been adding more physics, more fidelity, and more features into our codes lately, you’d see

1. its beginning to look like this
2. We’re suffering from the same problems.



So here’s where are research begins.

We know how computing started

We know how science went to SPMD programming

And we know how industry went to component programming.

What happens when you merge the two???

The End

Next: Intro to Components