Renovating a Legacy C++ Project

Alan Griffiths

alan@octopull.co.uk

Octopull Limited

www.octopull.co.uk

Who am I?

Alan Griffiths is a regular at the ACCU conference and has been developing software through many fashions in development processes, technologies, and programming languages.

During that time he's delivered working software and development processes, written contributions for magazines and books, spoken at a number of conferences and made many friends.

Firmly convinced that common sense is a rare and marketable commodity he's currently working as an independent through his company: Octopull Limited. (http://www.octopull.co.uk/)

A Rich C++ Legacy

The world is full of functionally rich, slow to build, hard to maintain, C++ systems. Some of these have been developed over time by many and varied hands.

They continue to exist because they provide valuable functionality to the organisations that own them.

To maximise their value it is necessary to provide interfaces to today's popular application development languages, and make it possible to continue to develop them in a responsive and effective manner.

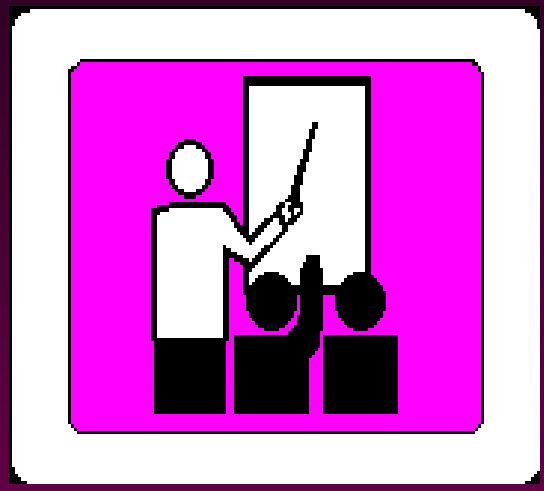
A Multi-language Paradigm

- As time goes by
  - Computers get more powerful, so...
  - We build bigger and more complex systems
  - We are using more powerful and specific tools
  - One programming language is not enough
- C++ is mixed with other languages
  - Java, C#, ...
  - Javascript, Python, Perl, ...

A C++ Legacy

This is the story of one such system, the problems it presented and the approach taken to addressing these problems.

While I'll have slides to guide discussion and tell this particular story, I also want to encourage the audience to share their experiences during the session.



What I'll talk about

- Overview of the system I worked on
- Interfacing to Java and C#
- Automating of Build and Release

System Overview

- C++ Quantitative Analytics library
  - Implements models used to calculate trade values
- Uses C++ to control...
  - Memory use
  - Memory layout
  - Processing order
  - Threading

System Overview

- Client applications
  - Java applications Windows & Linux
  - C# applications on Windows
  - An Excel plugin

System Overview

- Long build (especially on Windows)
- No clear public interface
- System tests (only)
  - long running
  - only on developer builds

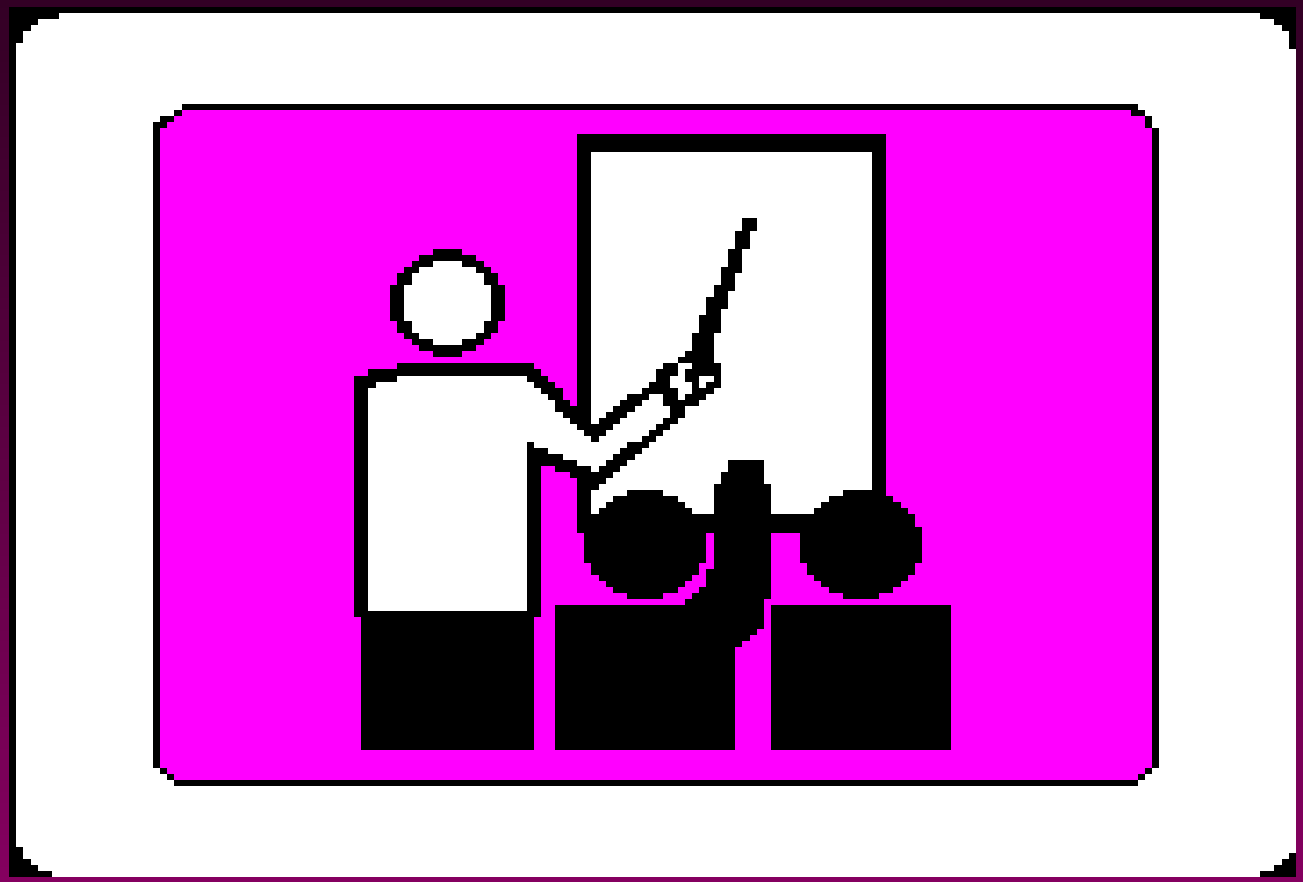
System Overview

- Developed on Windows

System Overview

- Deployed on
  - (32 bit) Windows
  - (32 bit) Linux

Comments & Questions?



The Legacy Interface

- Supply C++ headers and compiled libraries
  - All the .h files
  - .DLLs (and an .XLL) on Windows
  - A .so on Linux
- No clear public interface
  - (based on *all* the .h files)
    - Not obvious which changes break client code
    - Meets need of implementer, not user

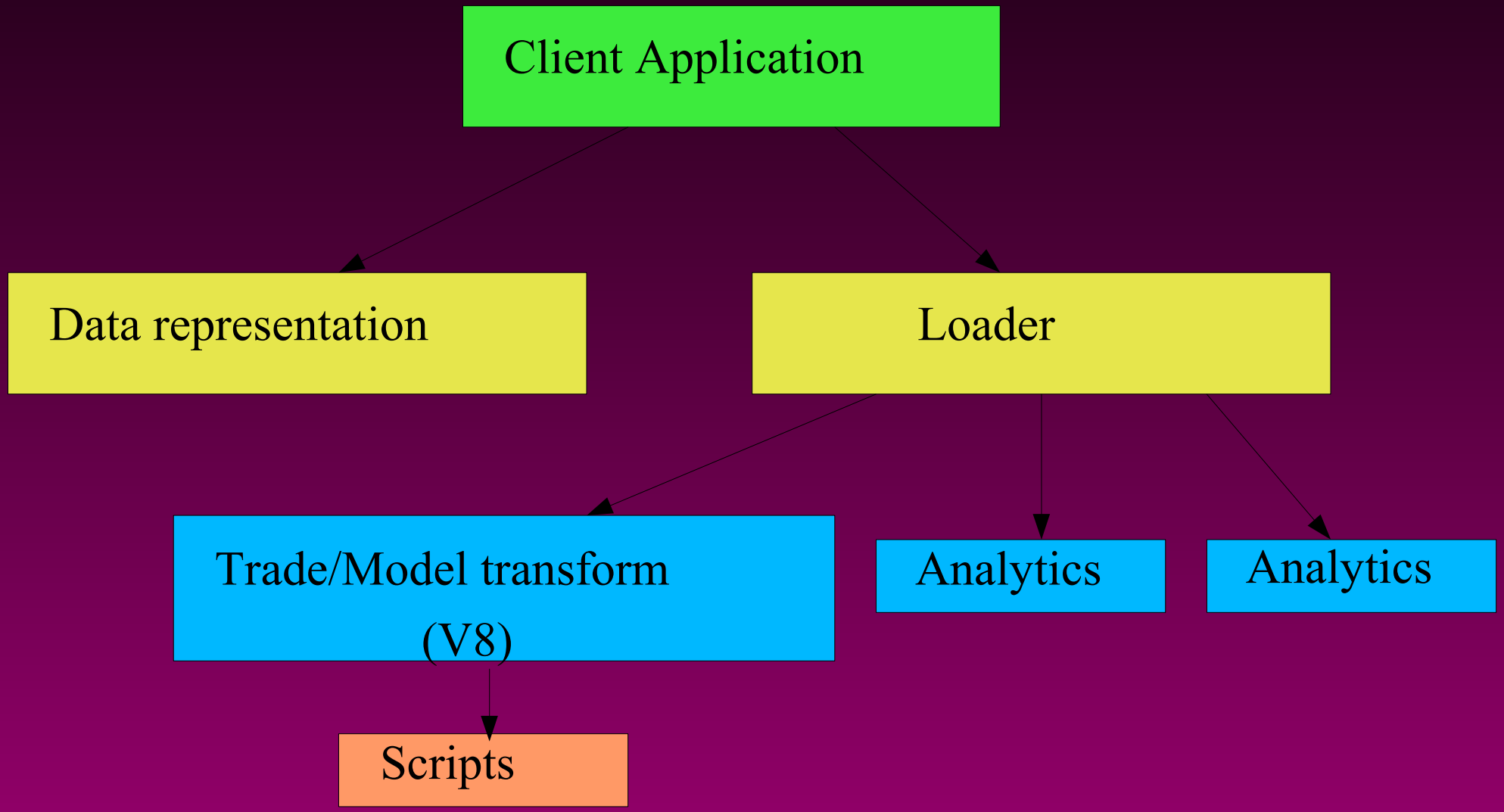
A SWIG Interface

- User applications are Java and C#
- Generate interface using SWIG
- SWIG code generation owned by a client team
- Not version controlled with analytics library
  - We compile and distribute but:
    - Can't change
    - Can't test
    - Can break

A New Interface

- Idiomatic Java/C# API
- Stable over time
- Based on user (not implementation) concepts
- Validated

New Architecture



Results

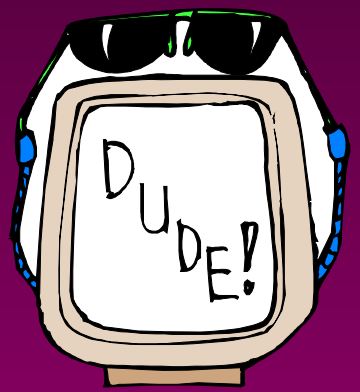
- Idiomatic C#
- Idiomatic Java

Results

- Stable over time
  - Some “tweaks” during first 6 months
  - A few naming changes later to conform to corporate initiative

Results

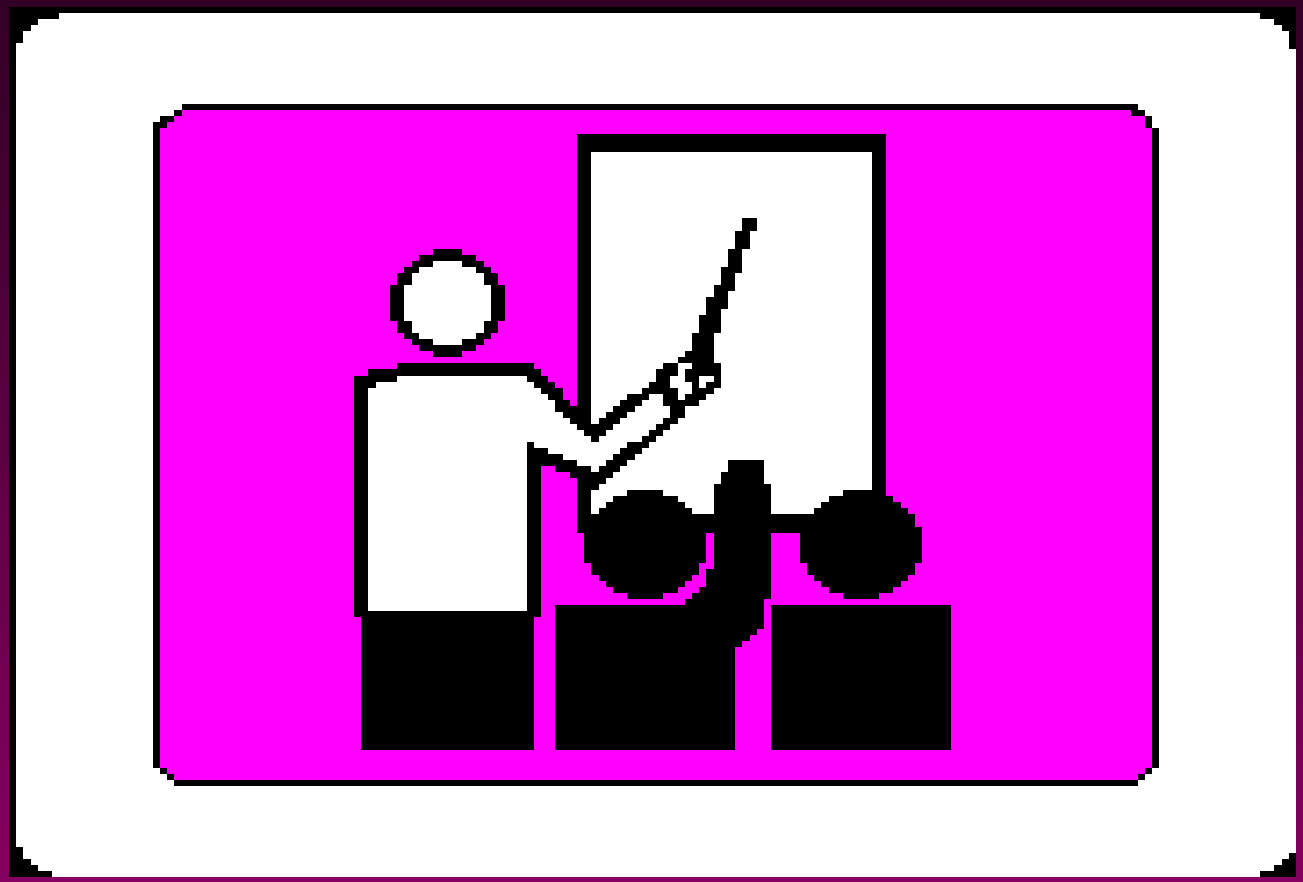
- Automated validation



Results

- Faster delivery by client applications
  - Before: 3 months
  - After: 3 weeks

Comments & Questions?



Building, Integration and Release

- Half “development” effort spent firefighting build and release problems

Continuous Integration

- Using CruiseControl
  - Building Release and Development branches
  - Building for multiple target platforms
  - Reporting on a chat channel
  - Not reporting on regression tests

Building

- Linux build failed a lot...
  - Makefile bug failed about half the builds
  - Developers got case wrong on #includes
  - Sometimes needed standard headers adding

Building

- Windows “Release” build depended on “Debug” build
  - Debug build wasn't used otherwise
  - “Debug(DLL)” used by developers
  - “Quantify” builds

Integration

- CruiseControl didn't control checkout and build
  - Delegates to agent running on target platform
    - Agent does checkout (of HEAD!)
  - Incorrect change reporting
  - Poor error reporting

Feedback from integration

- Slow
- “Noise” from spurious build failures (from buggy scripts)
- Misdirection from incorrect change recognition
- No reporting of test failures

Release

- The release process was
  - Multiple manual steps
    - pre-build and label
    - release build
    - stage
    - deploy
  - Fragile and error prone scripts
    - Not under CM

Slow progress on build

- Not only fixing build
  - Running releases
  - Developing new interface
- Poor structure to release scripts
  - Coupling prevents testing
  - Changes often “broke” in next release
- Gradually things improved
  - Makefile bug fixed
  - Debug and Quantify builds retired

Introducing TeamCity

- Build agents on multiple platforms
- Building the version CI reports on
- Artefact repository and traceability

Results: Easy to change

- Added regression and validation tests
- Changing compiler and library versions
- Setting up (and retiring) release branches
- Changing distribution targets

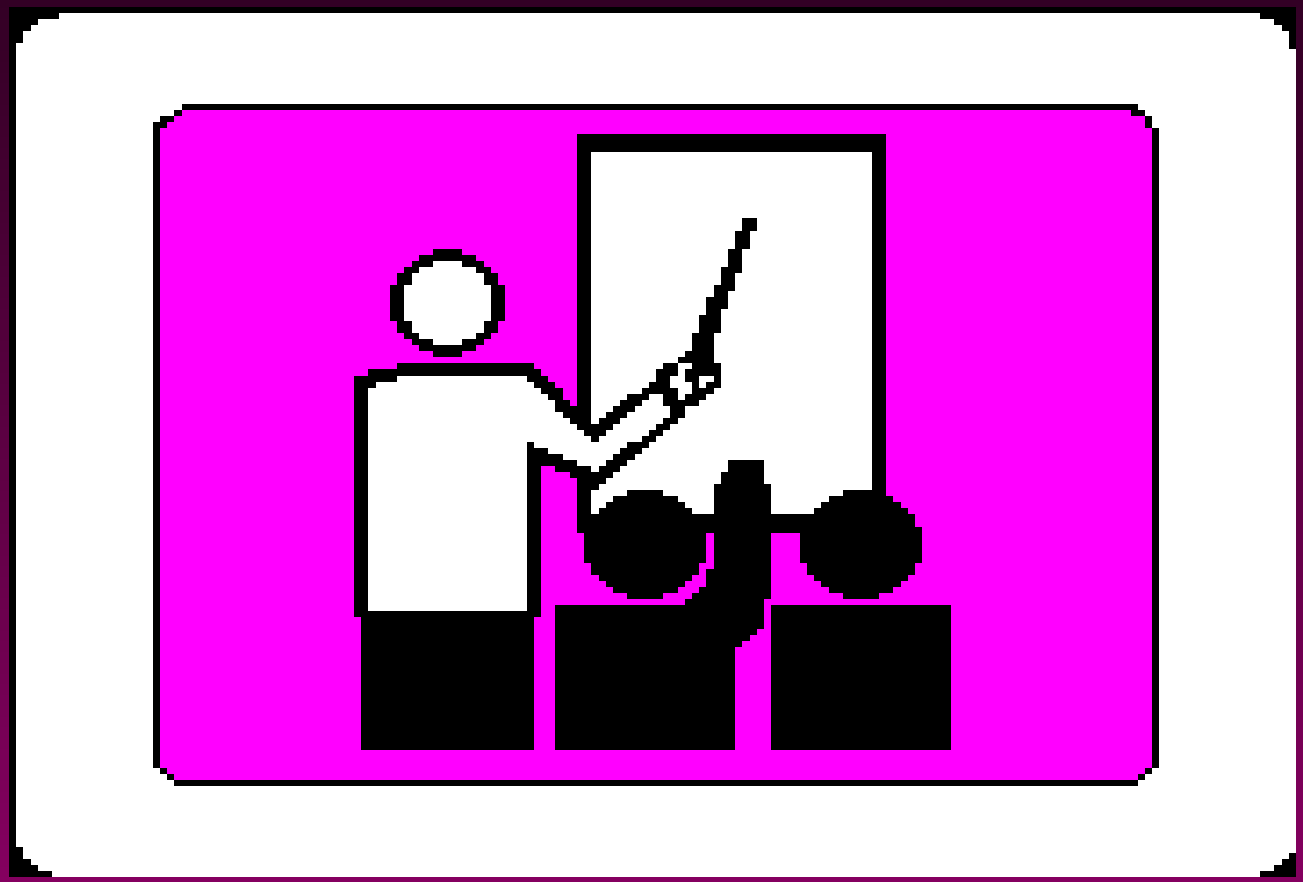
Results: Easy to Extend

- Multiple build agents
- New compiler versions
- 64 bit platforms

Results

- It took 6 months to get a maintainable build system
- Start:
  - could take a week and days of developer time to get a release out
- End:
  - From checkin to deployment took two hours and a few developer minutes on change bureaucracy

Comments & Questions?



“C++ for Quants”

- Head Quant set up lunchtime meetings
  - Run by volunteers (engineers or quants)
  - about engineering aspects of C++ code

Renovating a Legacy C++ Project

Alan Griffiths

alan@octopull.co.uk

Octopull Limited

www.octopull.co.uk

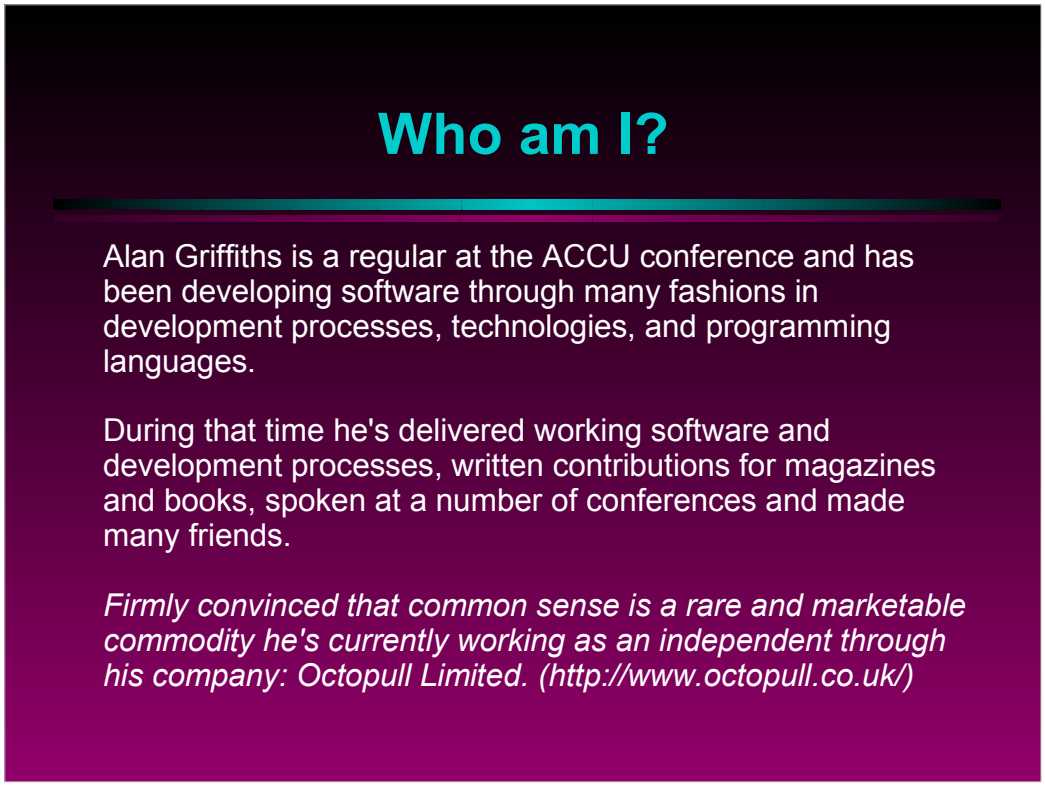
Renovating a Legacy C++ Project

Alan Griffiths

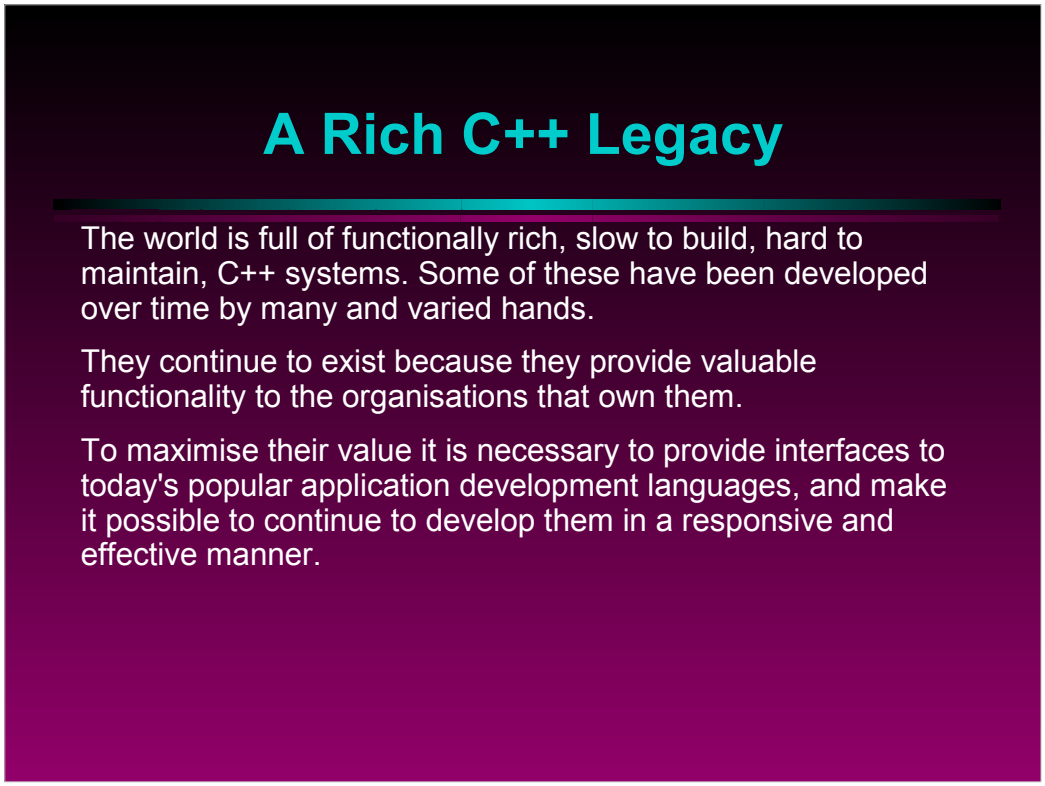
alan@octopull.co.uk

Octopull Limited

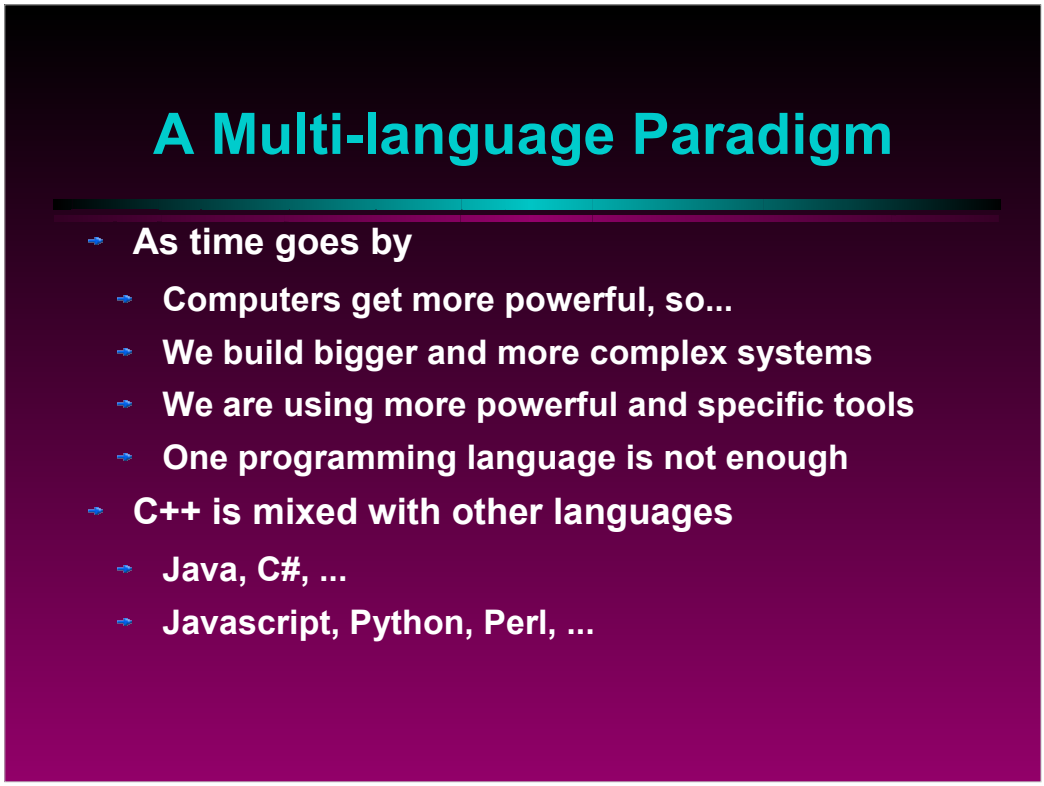
www.octopull.co.uk



I've been using C++ for a long time, at first because it was the principle language available for developing desktop applications, more recently either for non-technical reasons or because it provides better control over resources than other popular languages.



The project that I'm going to describe used C++ for a mixture of reasons – the non-technical reason was that the codebase has been developed over a couple of decades, originally in “C With Objects” but more recently, after a port, in C++ with man-centuries invested in the codebase a rewrite in a fashionable language would be hard to justify. The technical reasons for choosing C++ were the usual “control over resources” ones – principally CPU and memory.

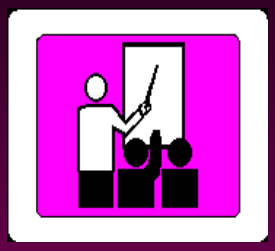


Over that time there has been an increasing need to work effectively with other languages more suited developing parts of the systems. In recent years the projects I've worked on have combined C++ with Java, C#, Python and Javascript.

A C++ Legacy

This is the story of one such system, the problems it presented and the approach taken to addressing these problems.

While I'll have slides to guide discussion and tell this particular story, I also want to encourage the audience to share their experiences during the session.



What I'll talk about

- Overview of the system I worked on
- Interfacing to Java and C#
- Automating of Build and Release

System Overview

- C++ Quantitative Analytics library
  - Implements models used to calculate trade values
- Uses C++ to control...
  - Memory use
  - Memory layout
  - Processing order
  - Threading

The code in question is a “Quantative Analytics Library” - it does the numeric analysis that underlies the valuation of the trades done by an investment bank. Among other things it builds multi-dimensional datastructures of largely floating point numbers and processes these on a number of threads – small changes to layout and processing order can have big effects on performance. (And the resulting numbers!) Using C++ does give some indeed control over this – while there are other plausible languages for this work C++ remains a popular choice for such code.

System Overview

- Client applications
  - Java applications Windows & Linux
  - C# applications on Windows
  - An Excel plugin

There are a lot of applications in different areas of the bank that make use of this library to value the trades they are making. Most of the Linux based users are using Java, and most of the Windows users are using C#.

System Overview

- Long build (especially on Windows)
- No clear public interface
- System tests (only)
  - long running
  - only on developer builds

The codebase is monolithic – highly coupled, incohesive and with no agreed “public” interface, but it does have a suite of tests that covers all the financial models supported in production and, at least in principle, any bug fixes do come with a corresponding test case. [These are system test, not unit tests.]

System Overview

- Developed on Windows

The “Quants” working on the analytics library work almost entirely in Windows.

System Overview

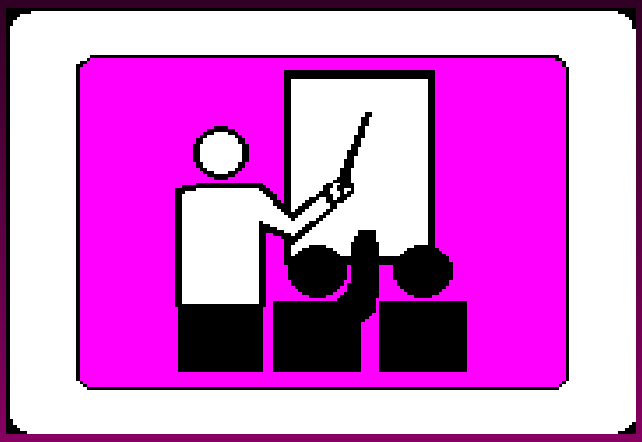
- Deployed on
  - (32 bit) Windows
  - (32 bit) Linux

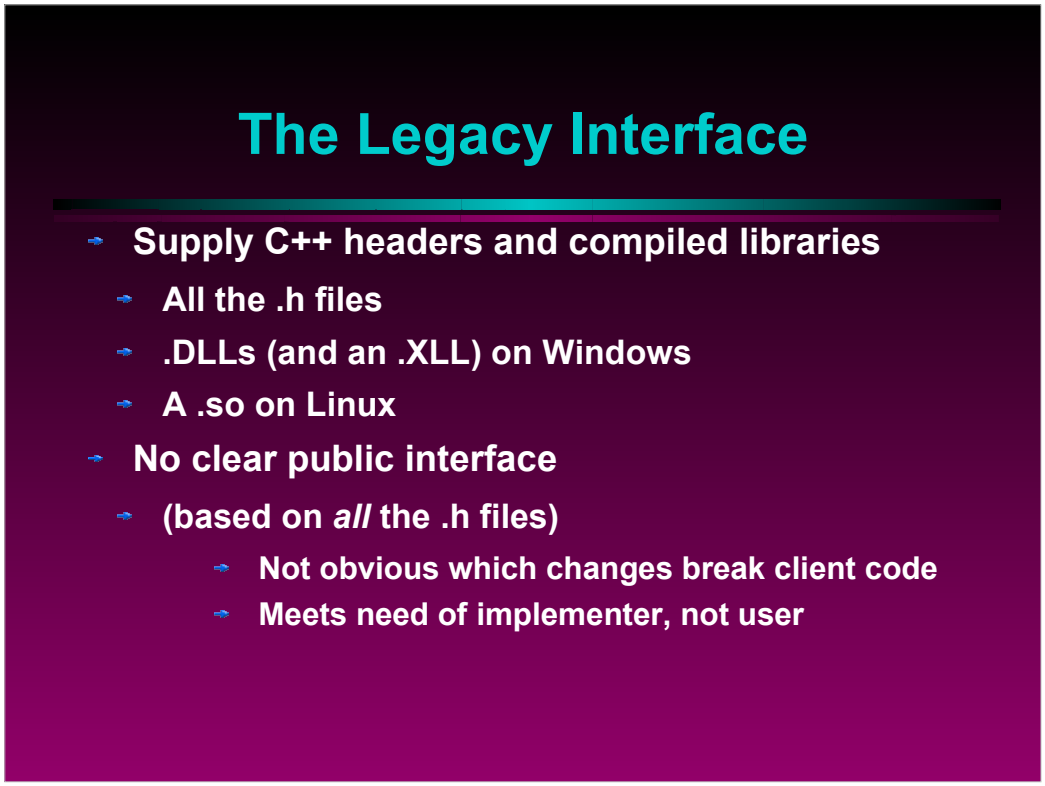
The library is supported on a range of platforms: initially 32 bit Windows and 32 bit Linux.

64 bit Linux is now supported for development use for roll out to production later this year.

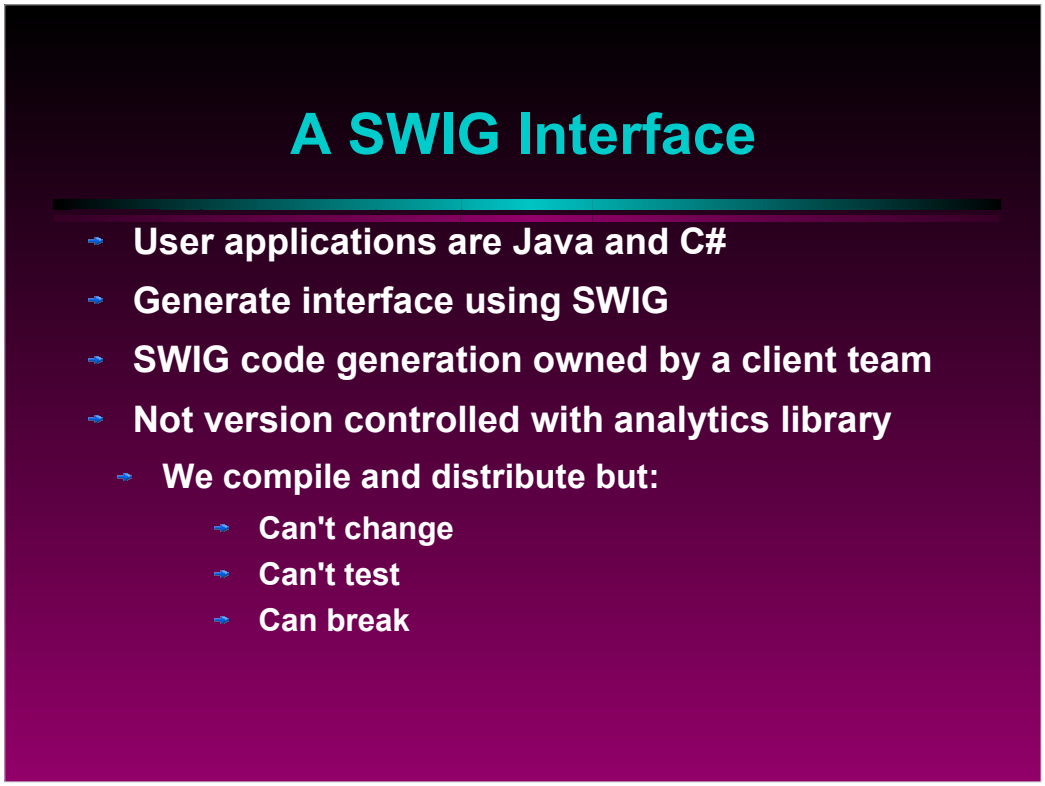
64 Windows is under development – to be supported next year.

Comments & Questions?





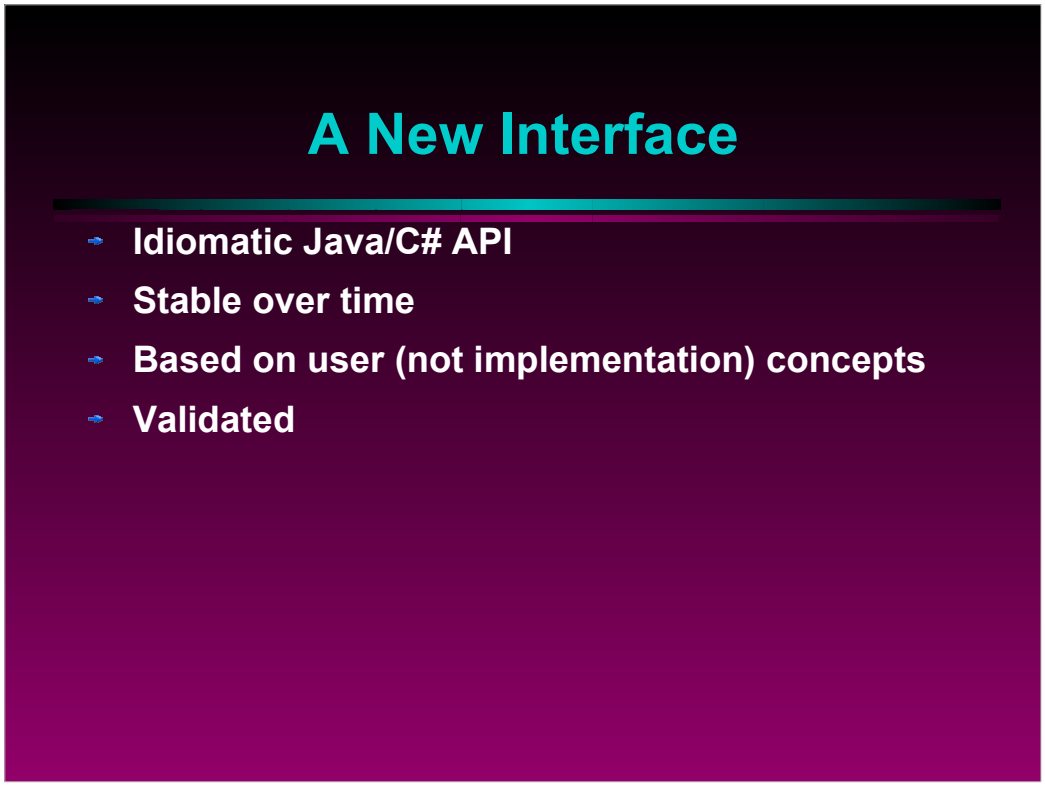
Historically the users have been given a set of libraries (.sos or .dlls) and all the header files extracted from the codebase. Neither the Java not C# users are particularly happy with this as the supported interface. Not only are things forever changing (because there is no agreed public interface), it is also far from clear how a particular type of trade should be valued. Each application team therefore has to do work to map from its own representation of trades to the corresponding analytic model for valuing it.



At some time in the past one such client group wrote a series of scripts to generate and build an interface to the library using SWIG. Other groups started using this interface and it is now shipped with the analytics library. This isn't ideal as, while there are multiple groups using this interface, there are no tests at all. Provided it compiles and links it will be shipped. When we started work the code wasn't even part of the main repository (it was a svn “extern” to a repository owned by the group who once employed the original author - we didn't have commit access).

The lack of ownership of this interface generating code was particularly problematic as the same code is pulled in by all the active branches. (There are typically a couple of branches in production and another in development – but this can increase occasionally.) The main problems occur if changes on one of the branches necessitate changes to the generating code – which, as it has lots of special case handling for particular methods and constructors, happens. (In particular SWIG isn't able to expose the C++ distinctions between const, references, pointers and smart pointers – this can, and does, lead to unintended duplication of method signatures. There are some sed scripts to remove problematic functions from the source code.)

One of the changes we made was to “adopt” this code into our repository so that we could fix problems for each of the active branches. This wasn't entirely satisfactory as we still had no tests or access to the client code to validate it against – the best we achieved was to ensure that it compiled and linked then wait for users to “shout”. (As we wanted people to move off the legacy interface we felt we'd “done enough” at this point.)

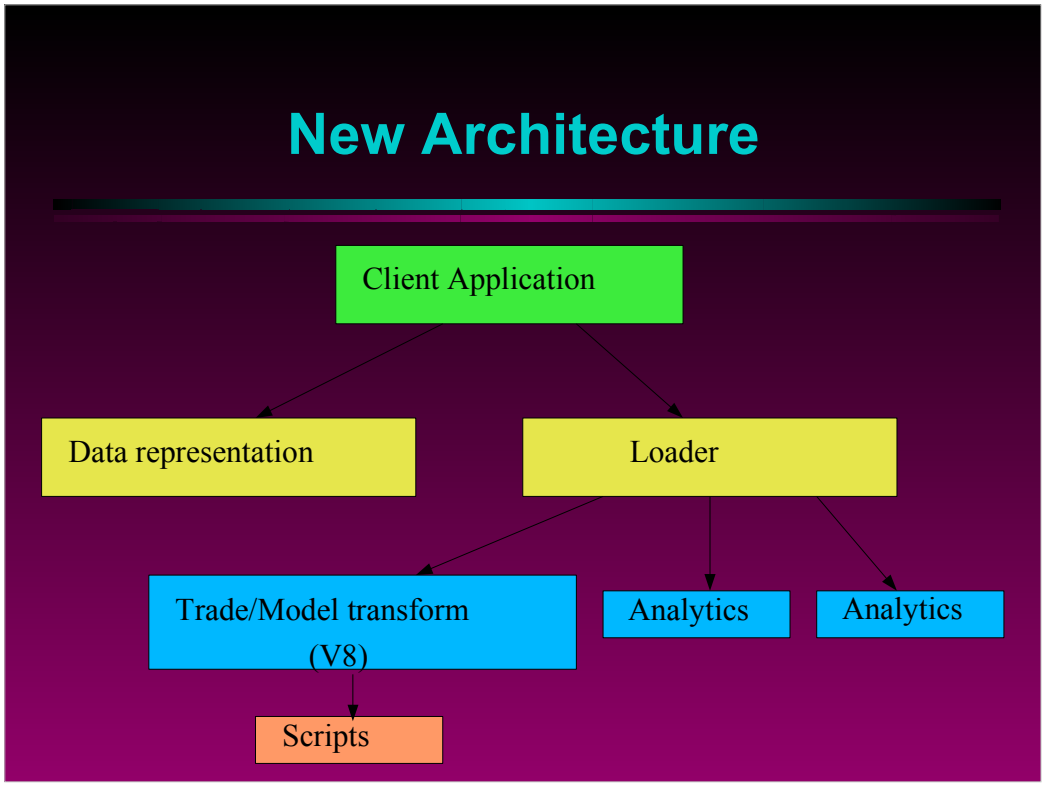


As mentioned earlier, the legacy interface caused problems for our users. The interface didn't reflect normal Java (or C#) conventions, wasn't stable and reflected the need of the implementer, not the user. Each client application's developers needed to understand not only the trades they were valuing but also the correct way to value them. In addition, they needed to get “sign-off” of the valuations being produced for each trade type they implemented.

With these issues to contend with it could take over three months to get a new type of trade and valuation into production. For competitive reasons the business wanted to move faster than this.

[[Prototype and issues with it – leaks, non-idiomatic interface, non-orthogonal functionality, etc.]]

To address this we built a new, more stable, public interface that directly supports Java and C# and incorporates the mapping between a trade definition and the valuation models. This should be much easier for client applications as the developers need only to present the trade in an agreed format and don't need to be concerned with the method of valuation.



This comprised a number of components:
One that supports a uniform data representation (this can be thought of as a subset of JSON – as that is it's serialised form);
Another that maps a trade representation to a model representation;
A third that uses the modelling library to value the model;
A fourth that manages all of this; and,
Native C# and Java APIs that provide access to all of this.

Much of this is implemented in C++, but there are obviously bits of C# and Java and the mappings between trades and models is implemented in Javascript.

Results

- Idiomatic C#
- Idiomatic Java

C# API designed using properties for the data representation and IDisposable for resources (so that “using” blocks correctly managed resources).

Java API designed using setters and getters and a “dispose()” method.

Results

- Stable over time
  - Some “tweaks” during first 6 months
  - A few naming changes later to conform to corporate initiative

The client interfaces have remained relatively stable over time, during the first six months there were binary interface changes, and through the first year there were tweaks to the trade definitions to approach a more uniform naming style and to co-ordinate with a global initiative to represent trade elements in the same way throughout the business. All of that is settling down and work now focusses on reflecting changes and enhancements to the valuation engine and providing mappings for new trade types.

Results

- Automated validation

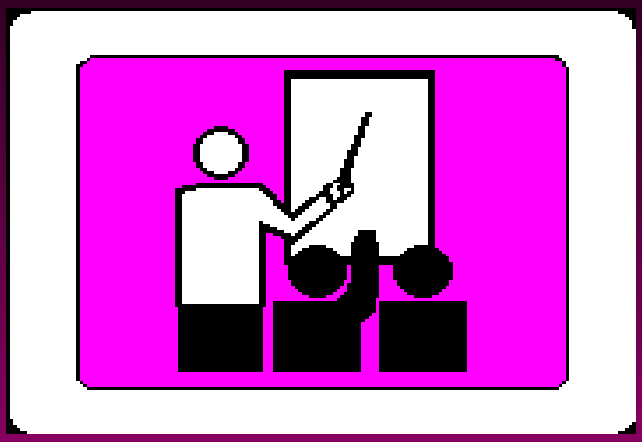
Naturally we introduced some “acceptance tests” for the supported trade types that ensured that they were validated correctly. This greatly simplified the task of application developers who now only need sign-off that they were presenting us with correct trade representations (our tests established the correctness of the results for all the application teams).

Results

- Faster delivery by client applications
  - Before: 3 months
  - After: 3 weeks

As a measure of success a new type of trade was implemented by a client application in three weeks instead of the three months that would have previously been required. They were also able to ditch several hundred thousand lines of code when migrating code to the new interface. (Although, as they were also removing a massive tangle of Spring that work must share some of the credit.)

Comments & Questions?



When I joined the project much of the effort was expended “firefighting” the build and release process. There were a number of problems:

Continuous Integration

- Using CruiseControl
  - Building Release and Development branches
  - Building for multiple target platforms
  - Reporting on a chat channel
  - Not reporting on regression tests

Building

- Linux build failed a lot...
  - Makefile bug failed about half the builds
  - Developers got case wrong on #includes
  - Sometimes needed standard headers adding

Even with good code the Linux build failed about half the time – this turned out to be a parallelism issue, one make rule created a directory, another wrote to it and there was no dependency between them. Some very fragmented makefiles (lots of includes) made this hard to spot.

Building

- Windows “Release” build depended on “Debug” build
  - Debug build wasn't used otherwise
  - “Debug(DLL)” used by developers
  - “Quantify” builds

The Windows “Release” build would fail if the Windows “Debug” build wasn't built first. Apart from this the “Debug” build wasn't used – the “Release” and “Debug” builds produced a single library, for developers there was a more useful “Debug(DLL)” build that allowed components to be worked on independently. Building all these configurations took a few hours.

Integration

- CruiseControl didn't control checkout and build
  - Delegates to agent running on target platform
    - Agent does checkout (of HEAD!)
  - Incorrect change reporting
  - Poor error reporting

CruiseControl was used to manage continuous integration but it didn't actually control the checkouts or build processes.

It took a while to figure out what was going on! What CC thought was a build was actually a shell script that wrote a token file to a shared directory – another shell script on the corresponding target platform looked for these tokens and checked out the current source and built it.

This then wrote the build results back to the share for the first script to pick up. The results were confusing as HEAD often changed between CC scanning the repository and the build script checking things out.

The idea of having an “agent” on another platform do the build is good – and centralises reporting – but this implementation left problems.

Feedback from integration

- Slow
- “Noise” from spurious build failures (from buggy scripts)
- Misdirection from incorrect change recognition
- No reporting of test failures

The speed of the build process was important as, with all the noise of meaningless build failures and the lack of clear feedback on who, if anyone, was responsible for breaking the build developers were not as alert as might be hoped to problems they caused with their checkins. HEAD was frequently broken so getting a release out as soon as possible after a good integration build was a way to mitigate problems.

Release

- The release process was
  - Multiple manual steps
    - pre-build and label
    - release build
    - stage
    - deploy
  - Fragile and error prone scripts
    - Not under CM

The release process worked in a similar way, except that there was a pre-release build that would label the release if it succeeded. Then the label was checked out and built to create the release binaries. Again, HEAD would often change after the build was requested. (And, as the build failed randomly half the time, it often took several attempts to build before a label was applied – with the codebase evolving all the time.)

Pushing the release binaries out to the development and production environments was also problematic. Apart from the inevitable organisational change control forms that needed to be completed, there was a sequence of scripts to be run in sequence. The various shell and perl scripts involved were fragile. They had hidden hardcoded interdependencies, poor error checking, needed to be run as specific accounts, and no documentation.

The scripts and Makefile targets used for developer, integration, pre-release and release builds were different and had subtle differences in behaviour. Several scripts coded the exact list of binaries to be distributed and their build locations – any time this changed something was likely to go wrong.

The release process was slow and failure prone – it copied the release to multiple development and production environments around the world (which could be offline or have space issues). With little error handling in the scripts it was necessary to inspect the output carefully.

While there was a test suite for validating the system it was only run on Windows and the results were not checked in the integration or release builds.

Slow progress on build

- Not only fixing build
  - Running releases
  - Developing new interface
- Poor structure to release scripts
  - Coupling prevents testing
  - Changes often “broke” in next release
- Gradually things improved
  - Makefile bug fixed
  - Debug and Quantify builds retired

Although these issues were a constant drain, fixing them was a background task – getting releases out was a full time job. On top of which there was the new library interface to be built. With a small team progress was slow.

Changing the build system was also time consuming - even with pairing to review changes, and trying to separate out bits that so that they could be tested independently, changes tended to break things in ways that were not detected until the next release was attempted.

But every time we fixed one of these problems life got easier. The race condition was the first fixed and made things a lot more predictable. Killing the dependency on “Debug” and eliminating the “Debug” build also sped up integration and release builds. (In later days the “Debug(DLL)” build was renamed “Debug” to keep things simple.)

Introducing TeamCity

- Build agents on multiple platforms
- Building the version CI reports on
- Artefact repository and traceability

The cure for this was, of course, to be able to build the version of the code that the CI tool was looking at. This could have been fixed with CruiseControl but, in practice, waited until we replaced CruiseControl with TeamCity. TeamCity has direct support for having "build agents" on a variety of platforms – which eliminated the scripting complexity used to achieve this with CruiseControl.

TeamCity also integrates an Ivy artefacts repository which allowed us to eliminate the pre-release and release builds – our new release process took the binaries directly from the integration build and tagged the corresponding source revision in the repository.

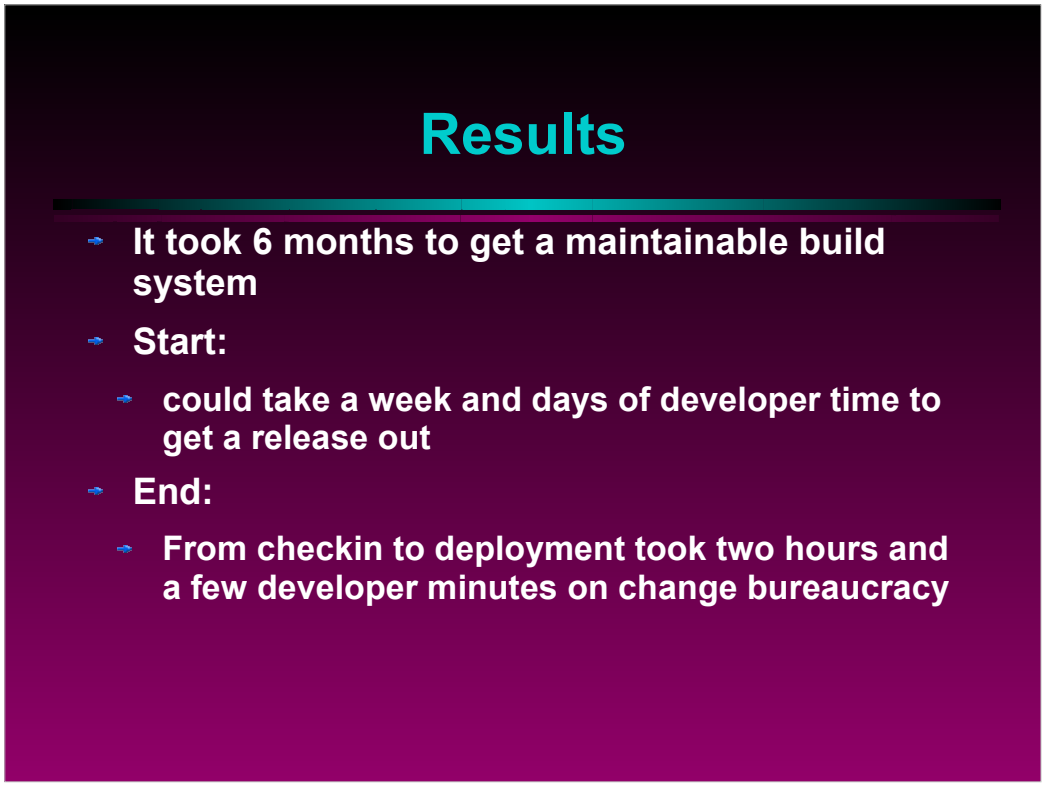
Results: Easy to change

- Added regression and validation tests
- Changing compiler and library versions
- Setting up (and retiring) release branches
- Changing distribution targets

Results: Easy to Extend

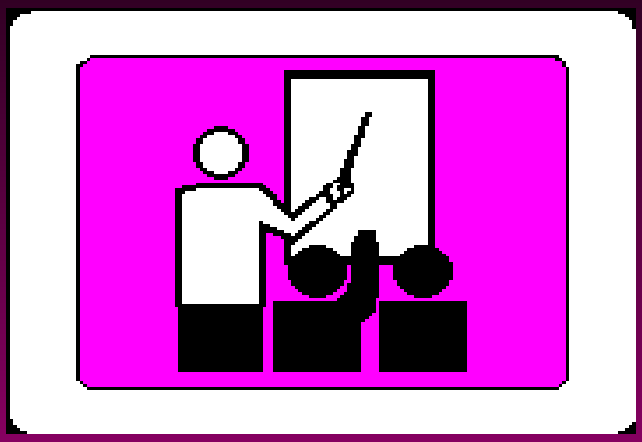
- Multiple build agents
- New compiler versions
- 64 bit platforms

Easy to add extra build configurations that use different compilers and/or target platforms



After the first six months we'd worked out way through the issues and life was a lot easier. If integration was “green” then a release could be requested of that code (and even if integration were broken, if the needed change was in the last successful build then that version could be released).

Comments & Questions?



“C++ for Quants”

- Head Quant set up lunchtime meetings
  - Run by volunteers (engineers or quants)
  - about engineering aspects of C++ code

Renovating a Legacy C++ Project

Alan Griffiths

alan@octopull.co.uk

Octopull Limited

www.octopull.co.uk