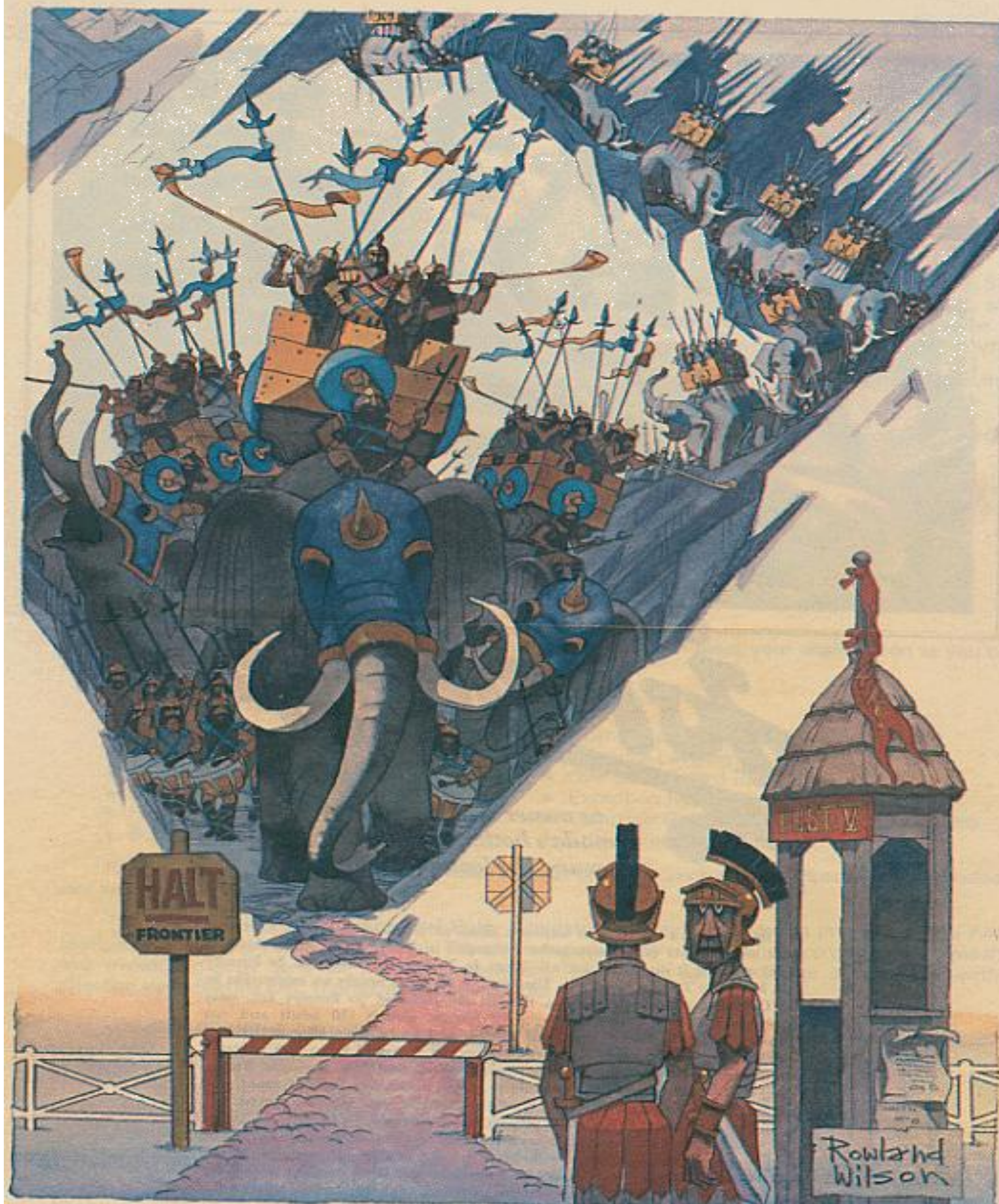
Software engineering



"Oh, oh . . . here comes trouble!"

CERN School of Computing

Exercises

1) Demonstration of a test framework
2) Practice debugging using a test framework
3) Demonstration of a profiling tool
4) Practice tuning a small application

If you want experience with CVS, we’ve got optional exercises:

A) Simple use of CVSB) More advanced CVS, showing how conflicts are handled

If you want some more practice with performance tuning, we've got two optional exercises:

5) Understanding, updating and tuning a larger application
6) Tuning a sample RSA encryption/decryption application

7) Simple release activities with CMT
8) Releasing code changes with CMT
9) Managing configuration conflicts
10) Project - Joint Development

Instruction sheets are available via web browser at

file:/home/jake/CSC/index.html

Exercise 1 - testing "SumPrimes"

Should "len" be included or not?

Its OK for a prime number to be divisible by one

```
int sumPrimes(int len) {
    int sum = 0;
    for ( int i=1; i < len; i++ ) {  // loop over possible primes
        bool prime = true;
        for (int j=1; j < 10; j++) { // loop over possible factors
            if (i % j == 0) prime = false;
        }
        if (prime) sum += i;
    }
    return sum;
}
```

If you divide a number by itself, the remainder is zero

Lesson 1: Its not easy to understand somebody else's code

- Assumptions, reasons are hard to see

  "Is one a prime number?"

  Test defines the behavior!! assertTrue(sumPrimes(1)==1)

Lesson 2: Better structure would have helped

- Separate "isPrime" from counting loop to allow separate understanding
- Make the algorithm for checking prime even clearer

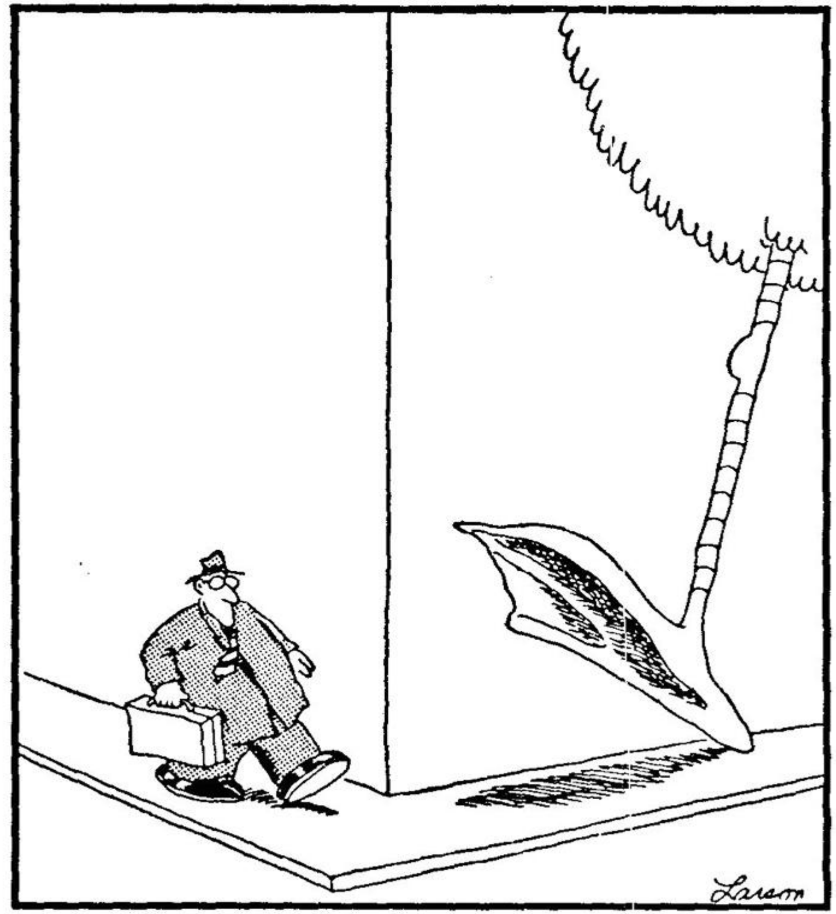
Exercise 2 - isCube, isSquare, et al

New bugs:

- Just introduced
- Newly discovered in another area
- Newly understood to be bugs

Too many possibilities, how do you keep track?

This is why large projects get harder as you go along!



Harold would have been on his guard, but he thought the old gypsy woman was speaking figuratively.

The life time of HEP software

R. Brun

CERN School of Computing

Software is a long-term commitment

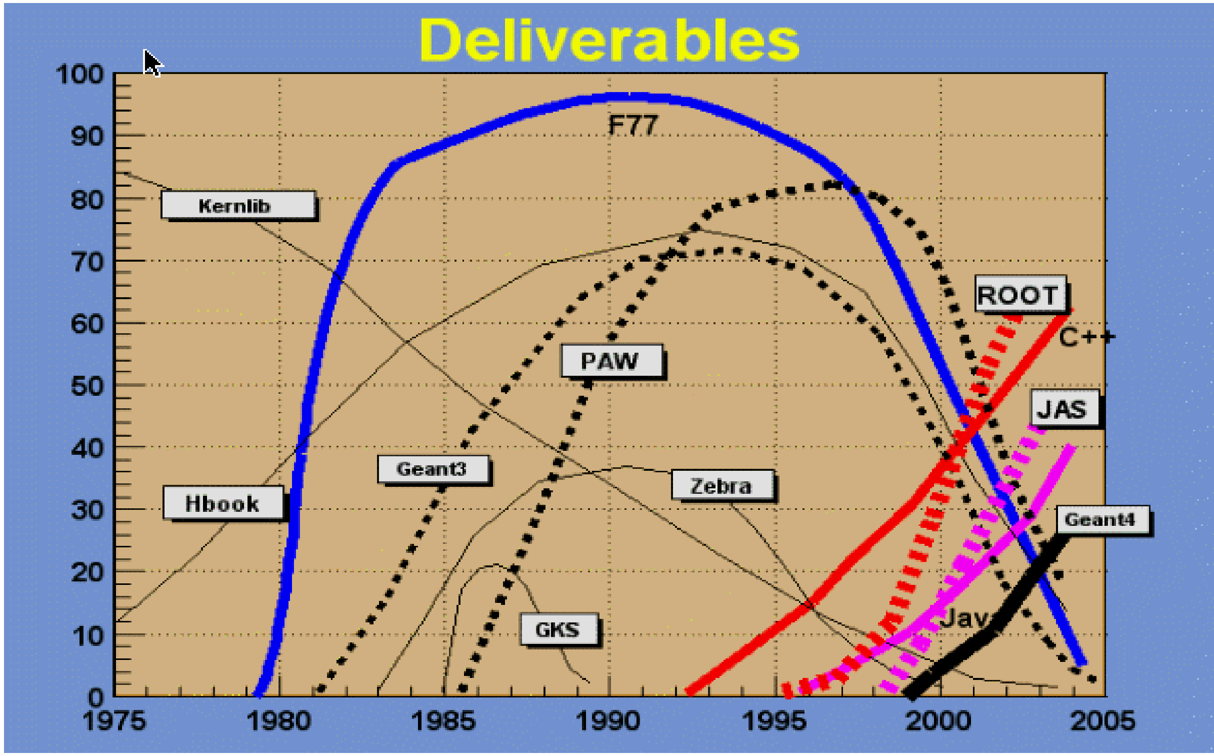
Users like stable and maintained systems

Vote with their feet

It takes time to develop a new system

- Geant3 6+ yrs 3 people 300 KLOCs
- PAW 6+ yrs 5 people 300
- Zebra 4+ yrs 2 people 100
- ROOT 5* yrs 3 people 630
- Working system after 1 year.

Real work is after that !!



Many releases of the software are needed over its lifetime to fix bugs, add new features, support new platforms etc

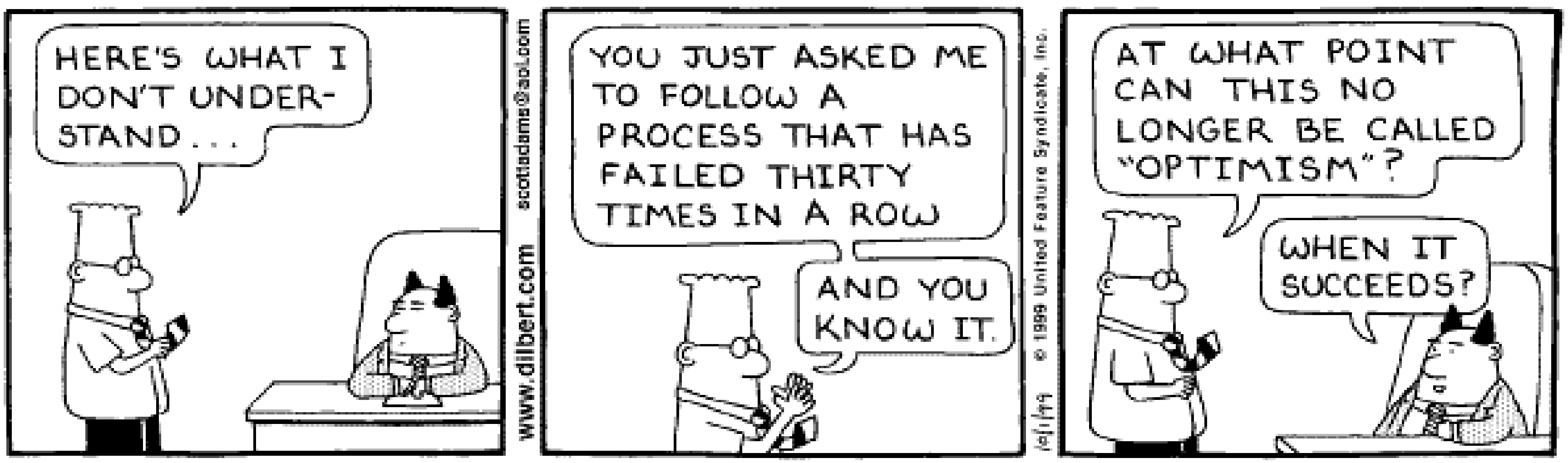
How do we cope?

CERN School of Computing

We try to find a way of working that leads to success

- **We create a "process" for building systems**
- **We devise methods of communicating and record keeping: "models"**
- **We use the best tools & methods we can lay our hands on**

And we engage in denial:



Can't technology save us?

We've built a series of ever-larger tools to handle large code projects:

CVS for controlling and versioning code

SRT for building "releases" of systems

CMT for "configuration management"

But we struggle against three forces:

- We're always building bigger & more difficult systems
- We're always building bigger & more difficult collaborations
- And we're the same old people

Net effect: We're always pushing the boundary of what we can do

Stupidity got us into this mess; why can't it get us out? - Will Rogers

CVS Source Code Management

Maintains a repository of text files

- Allows users to check in and check out changed text
- Old code remains available
  - Each checked-in change defines a new revision
  - You can retrieve, ask for differences with any of them
- Revisions can be tagged for easy reference

Anybody can get a specific set of source code file versions

Collaboration can use “tags” to control software consistency

Big advantage: checkout is not exclusive

- More than one developer can have the same file checked out
- Developers can control their own use of the code for read, write
- Changes can come from multiple sources
- CVS handles (most) of the conflict resolution

Key tool for large collaborations!

- But can also be an important tool for individuals

Why isn’t CVS enough?

CVS let’s me “check out” complete source code. Then just compile!

- **Works great for small projects**
- **But runs into several levels of scaling problems**

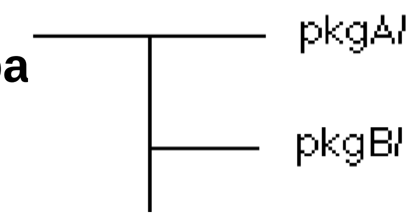
Want to attach to external code

- **We don’t write everything (though tempted)**
- **Sometimes don’t get source for external code**
- **Need some way to connect to specific external libraries:**

  Both specific product, and a specific version of that product

Want to separate code into multiple parts

- **So people/institutions can take responsibility for pa**
- **But software has cross-connections**
- **Need structure that works for both**



And still need to be able to build the code

Handling complicated builds

CERN School of Computing

Multiple "packages" require cross connects while compiling

- Typing the compile command gets boring fast

```
g++ -c -I"/afs/cern.ch/user/s/scherzer/public/1001/InstallArea/include/PixelDigitization"
-I"/afs/cern.ch/user/s/scherzer/public/1001/InstallArea/include/SiDigitization"
-I"/afs/cern.ch/atlas/software/dist/10.0.1/InstallArea/include/InDetSimEvent"
-I"/afs/cern.ch/atlas/software/dist/10.0.1/InstallArea/include/HitManagement"
-I"/afs/cern.ch/atlas/software/dist/10.0.1/InstallArea/include/TestTools"
-I"/afs/cern.ch/atlas/software/dist/10.0.1/InstallArea/include/TestPolicy"
-I"/afs/cern.ch/atlas/offline/external/Gaudi/0.14.6.14-pool201/GaudiKernel/v15r7p4"
-I"/afs/cern.ch/sw/lcg/external/clhep/1.8.2.1-atlas/slc3_ia32_gcc323/include"
-I"/afs/cern.ch/sw/lcg/external/Boost/1.31.0/slc3_ia32_gcc323/include/boost-1_31"
-I"/afs/cern.ch/sw/lcg/external/cernlib/2003/slc3_ia32_gcc323/include"  -O2 -pthread
-D_GNU_SOURCE -pthread -pipe -ansi -pedantic -W -Wall -Wwrite-strings -Woverloaded-virtual
-Wno-long-long -fPIC -march=pentium -mcpu=pentium -pedantic-errors -ftemplate-depth-25
-ftemplate-depth-99  -DHAVE_ITERATOR -DHAVE_NEW_IOSTREAMS  -D_GNU_SOURCE
-o PixelDigitization.o  -DEFL_DEBUG=0 -DHAVE_PRETTY_FUNCTION -DHAVE_LONG_LONG
-DHAVE_BOOL -DHAVE_EXPLICIT -DHAVE_MUTABLE -DHAVE_SIGNED -DHAVE_TYPENAME
-DHAVE_NEW_STYLE_CASTS -DHAVE_DYNAMIC_CAST -DHAVE_TYPEID
-DHAVE_ANSI_TEMPLATE_INSTANTIATION -DHAVE_CXX_STDC_HEADERS '
-DPACKAGE_VERSION="PixelDigitization-00-05-16"'  -DNDEBUG  -DCLHEP_MAX_MIN_DEFINED
-DCLHEP_ABS_DEFINED -DCLHEP_SQR_DEFINED     ../src/PixelDigitization.cxx
```

Build tools: "make", "Ant", etc

- Manually create a "makefile" that forwards include options to the compiler

  g++ -IpkgA -IpkgB

- Lets you adapt to various internal structures

  g++ -IpkgA -IpkgB/include -IpkgC/headers

- Also lets you add other options to control debugging, etc

But size keeps getting in the way

CERN School of Computing

BaBar (offline production code only):

- 430 packages
- 17,000 files
- 7 million lines of source

Some of these are large “for historical reasons”

But that’s true of just about any project

CVS checkout: 37 minutes

Build from scratch: 9 hours

Spread across multiple production machines; never did complete on laptop

“gmake” with one change: about 5-15 minutes to think about dependencies

And I don’t even want to think about the size of a monolithic Makefile

And everybody will need multiple copies…

Old ones, new ones, …

“But I just want to run the program!”

"Release Systems" are built to deal with this

CERN School of Computing

Key capabilities:

- Partial builds, including the case of "just run it"

- Ensuring consistency among the parts

Key concepts:

- "Release": labeled, consistent build of the entire system

- "Package version": name for a particular set of contents
  - The purpose of development is to change the contents of packages!
  - Helpful to have these be independent, so people can work independently

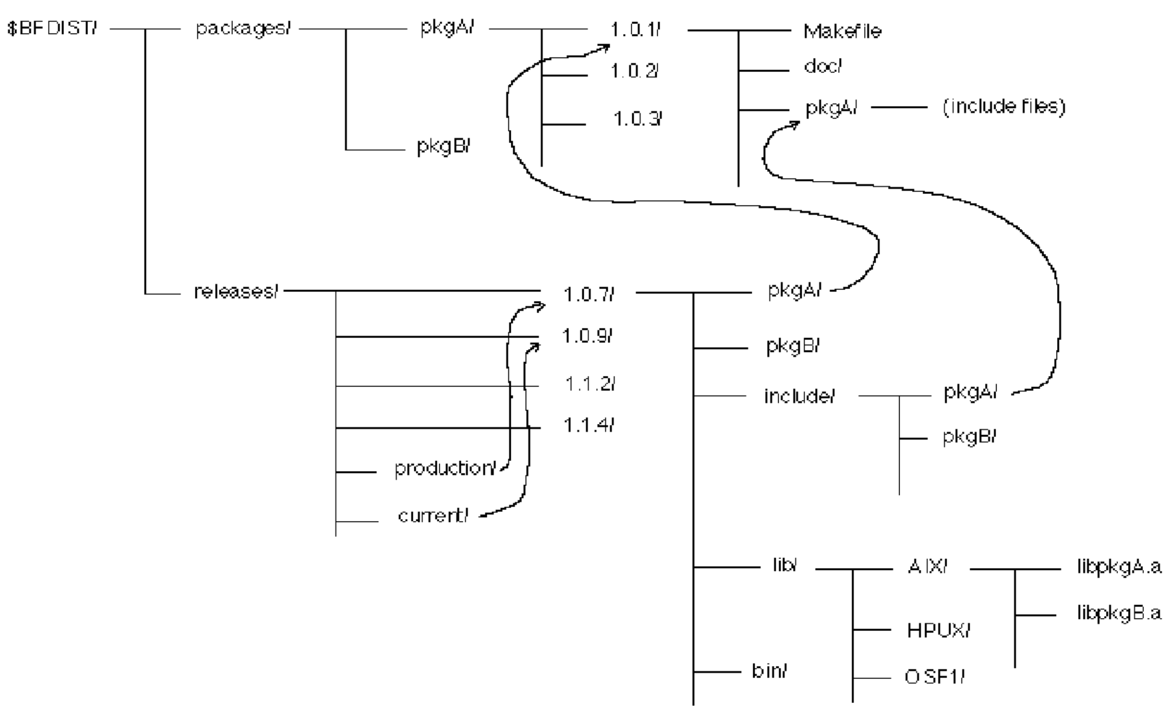
- "Architecture": A particular type of computer
  - hardware, software, even location

Simple Example: SRT (SoftRelTools)

CERN School of Computing

- Allows a build to mix existing (shared) and individual parts
  - Check out some packages & built just those
  - Pre-built libraries, include files, etc are matched in "versions"

- Set of shell scripts and Makefile fragments
  - Work within a particular directory structure



CERN School of Computing

Typical use:

- Create an area for your own work
  - Specify the production release you want as context
- Checkout source for the package(s) you want to edit
  - Specify which contents
  - Typically either the one from the context, or the latest
- Compile, test, debug, edit, repeat

- Eventually, you've made progress, and want to share it
  - Check changes back in
    - Now they're safe, and colleagues can get changes
  - Tag repository
    - So you can tell your colleagues how to get these as stable content
  - Make part of next "production" release
    - Typically a "package coordinator" role to decide about this

- These steps do <u>not</u> have to happen quickly, all at once, or by same person
  - Biggest differences between collaborations occur here

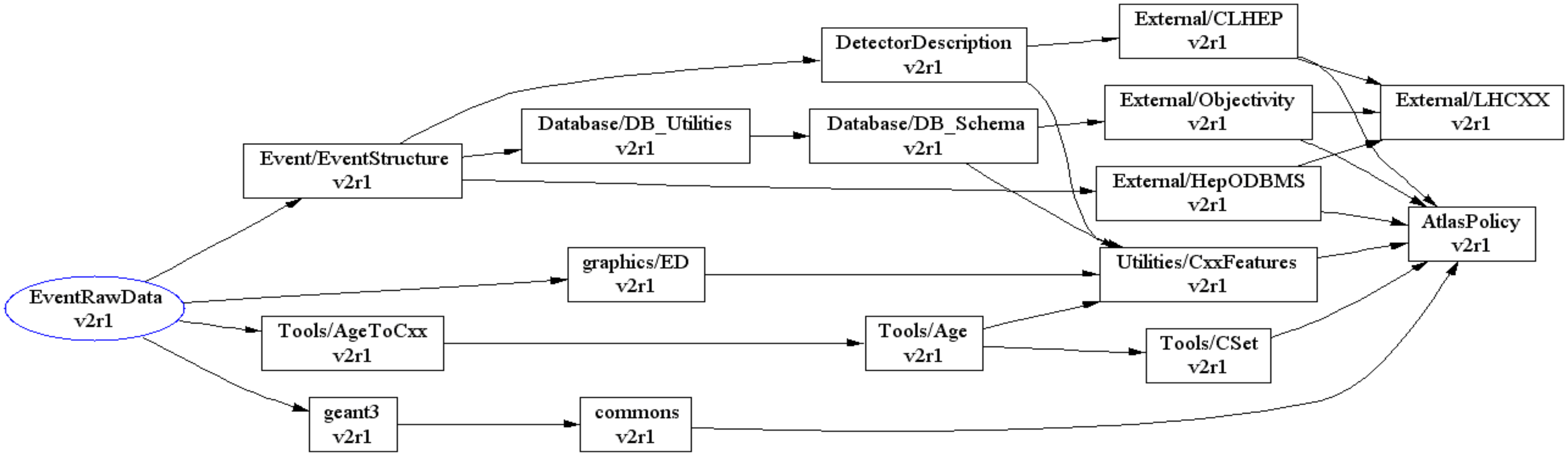
What else do we want from a release system?

- Better support of development
  - Not just building complete versions
    - Also want to build & run test scaffolds
  - More complicated package, release structures
    - Not just a flat set of co-equal packages with no substructure
  - Including enough flexibility to develop release tool itself

- Help distributing the workload
  - SRT spread parts of load across lots of package coordinators
  - But somebody still had to pull the production releases together
    - "Did you run your unit tests?"
    - If I update pkgA to V01-00-03, will pkgB V02-01-00 still work?

- Help ensuring consistency
    - If I update pkgA to V01-00-03, will pkgB V02-01-00 still work?

“Consistency”



Software strongly depends on other software

- **Usually managed at the package level**

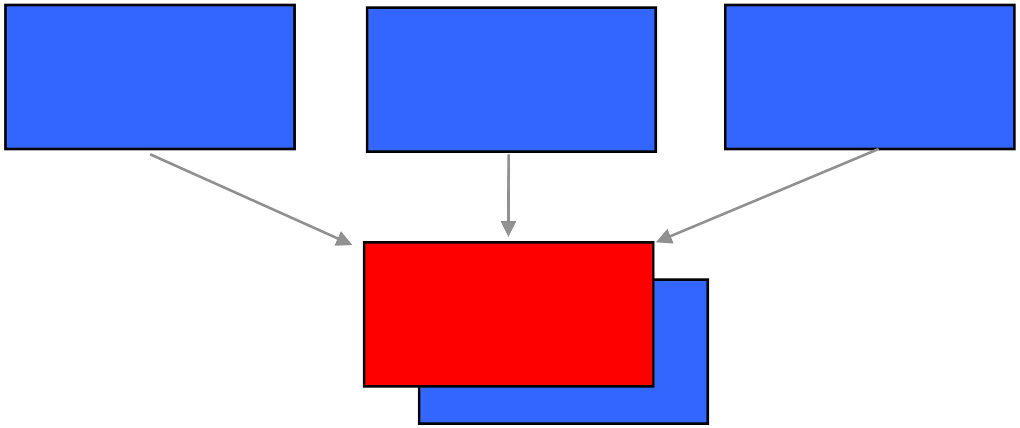
  (This can result in lots of packages, as you subdivide over and over)

- **Expresses how changes in one piece can drive changes in another**

Robert Martin's "open/closed" principle

Some parts of the code need to be "stable"

Other parts are being continually developed

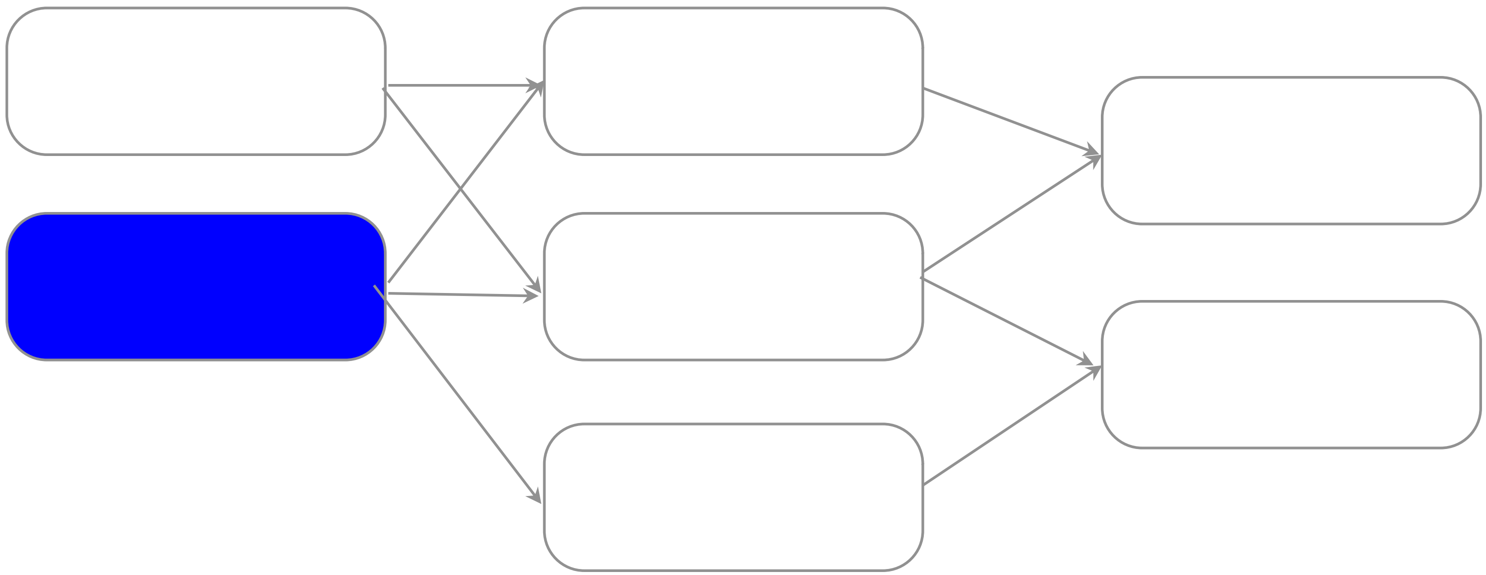


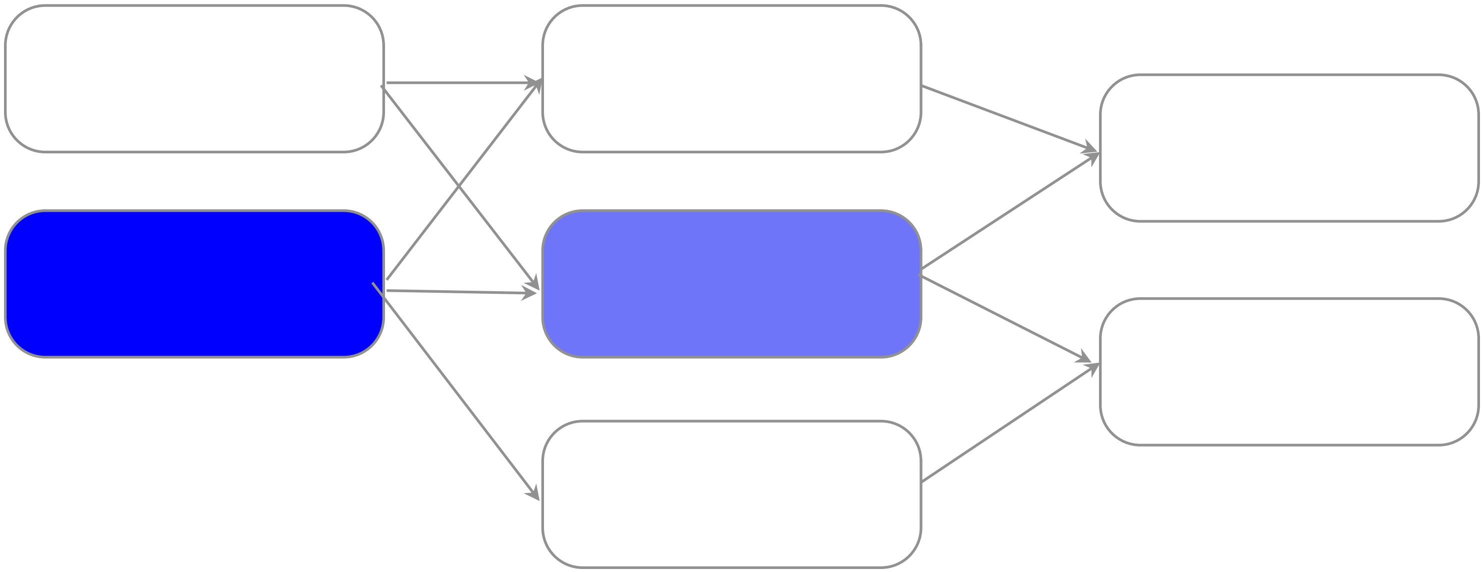
One solution: Separate stable interfaces from evolving implementations

But even stable interfaces have to change sometimes

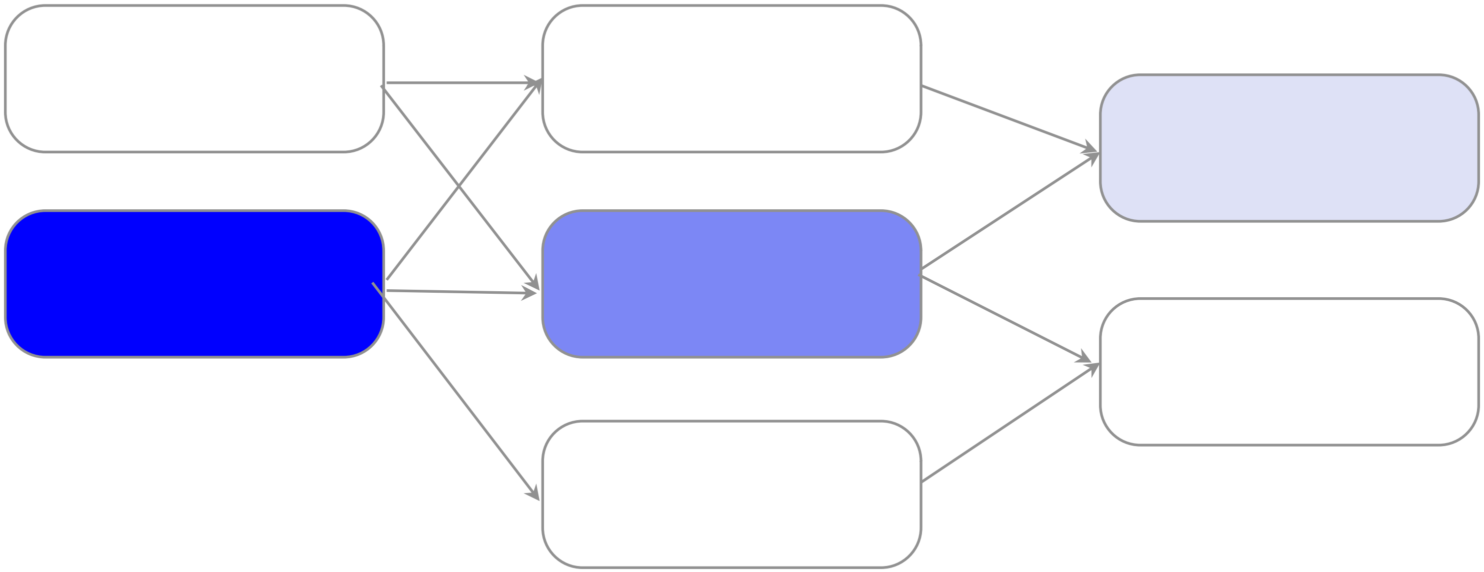
And you also need tools for handling dependence on external code, compiler/OS differences, location differences, etc

How change propagates through dependencies

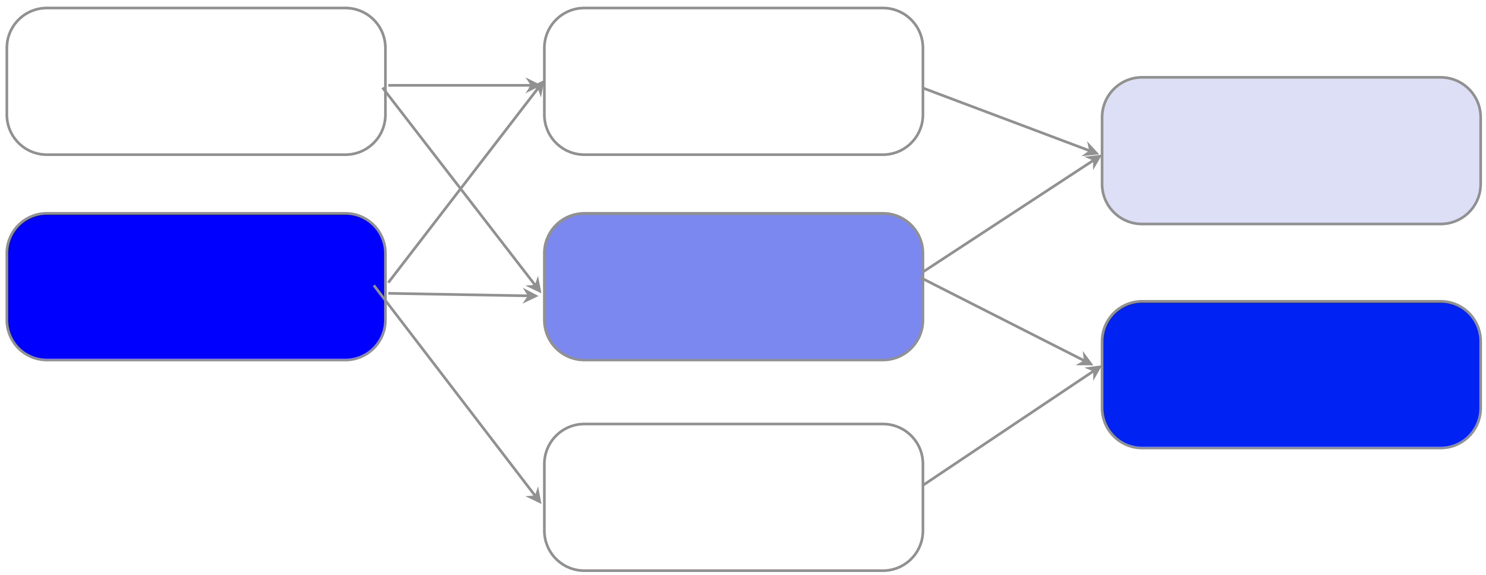




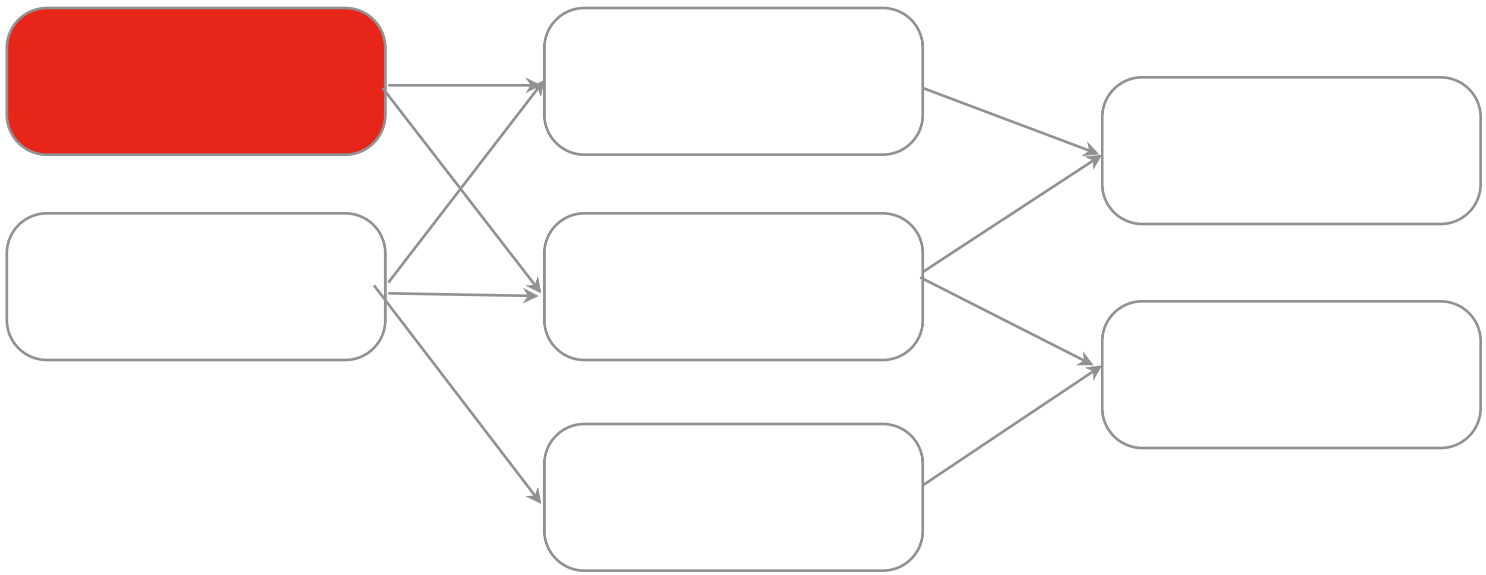
CERN School of Computing

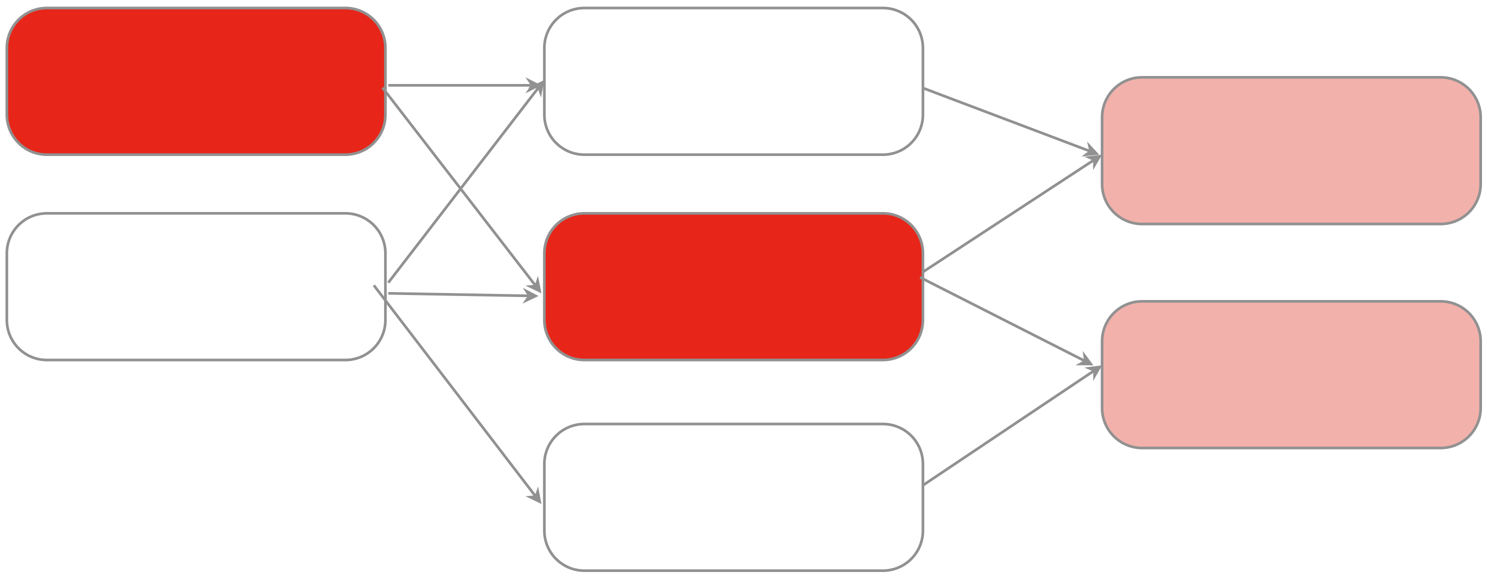


Changes don't always stay small

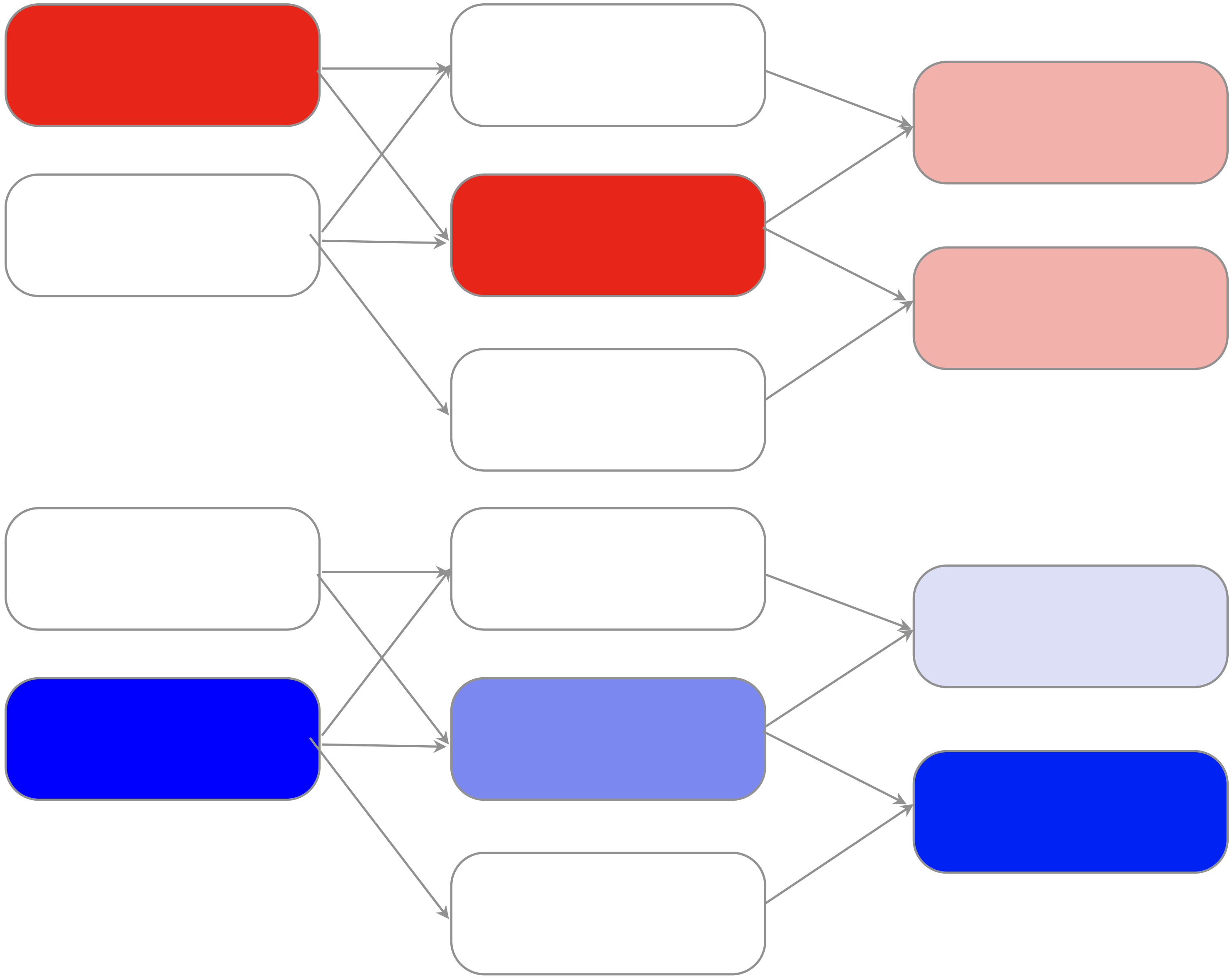


Another change:

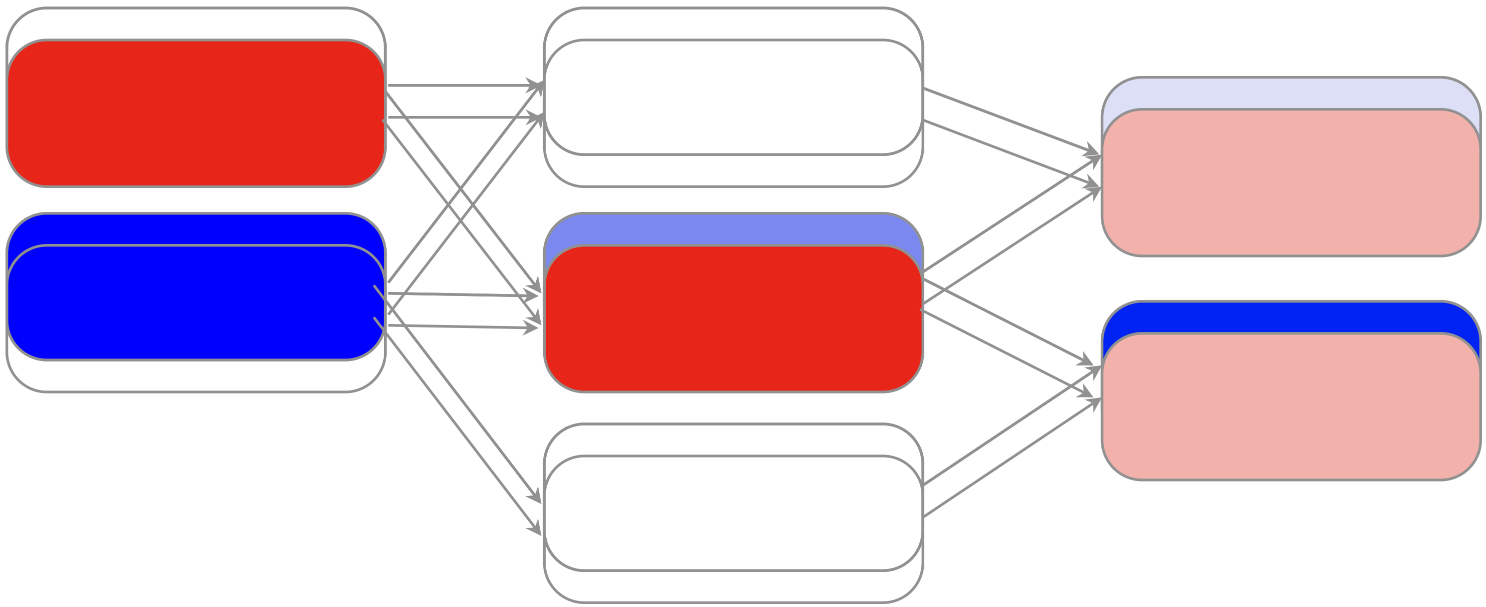




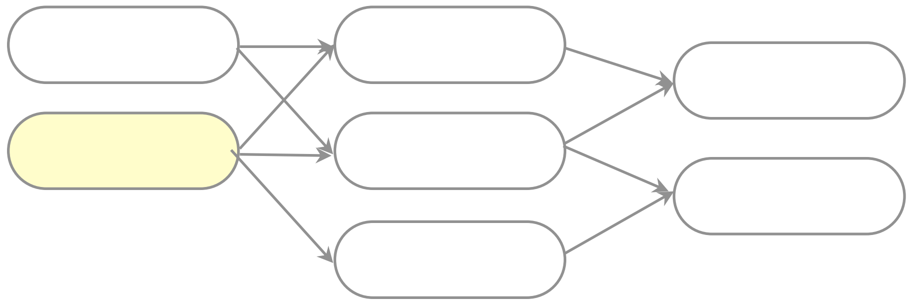
Change management

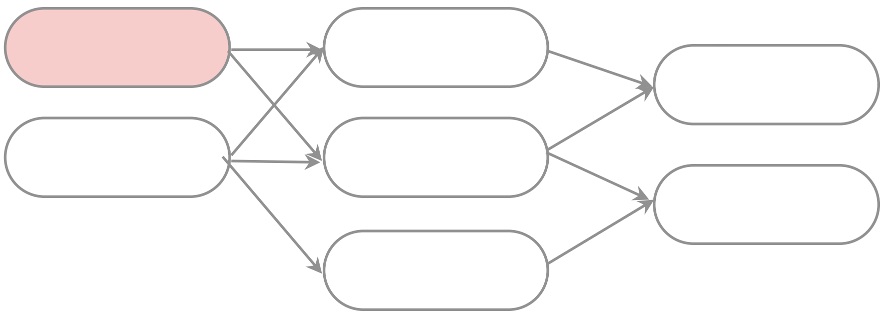


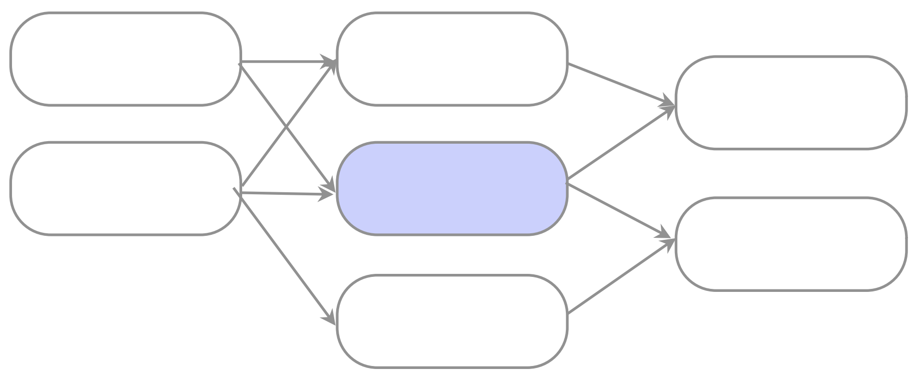
Change management

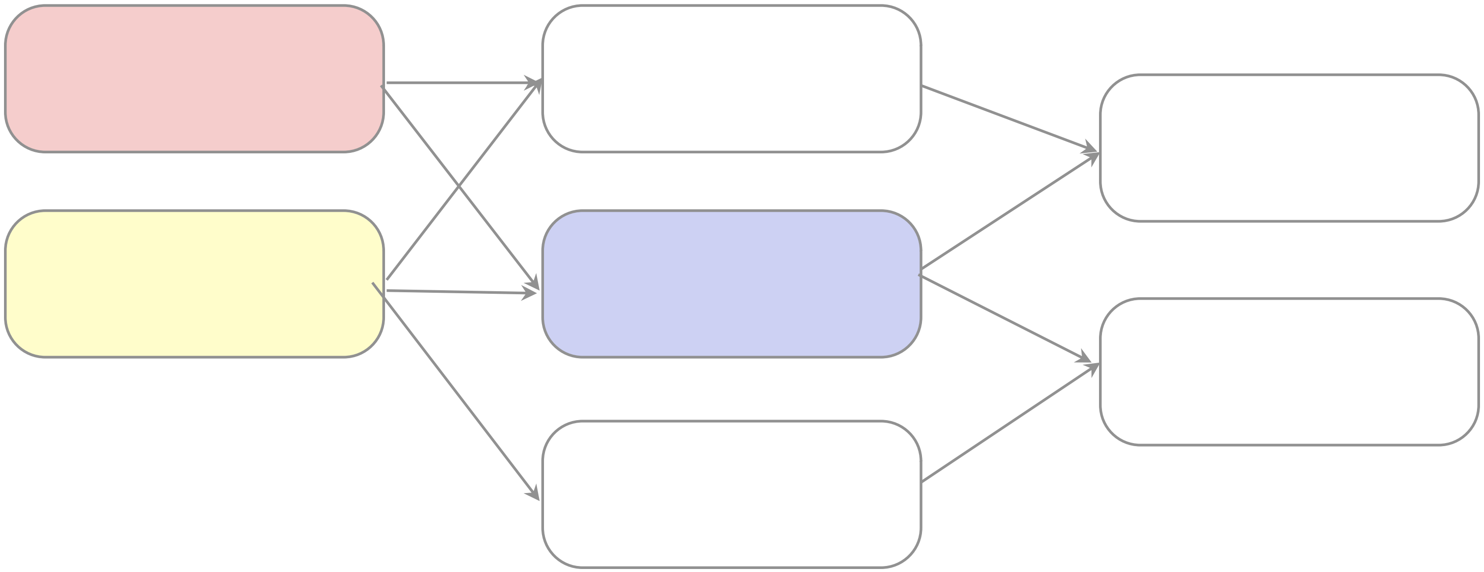


The worst case









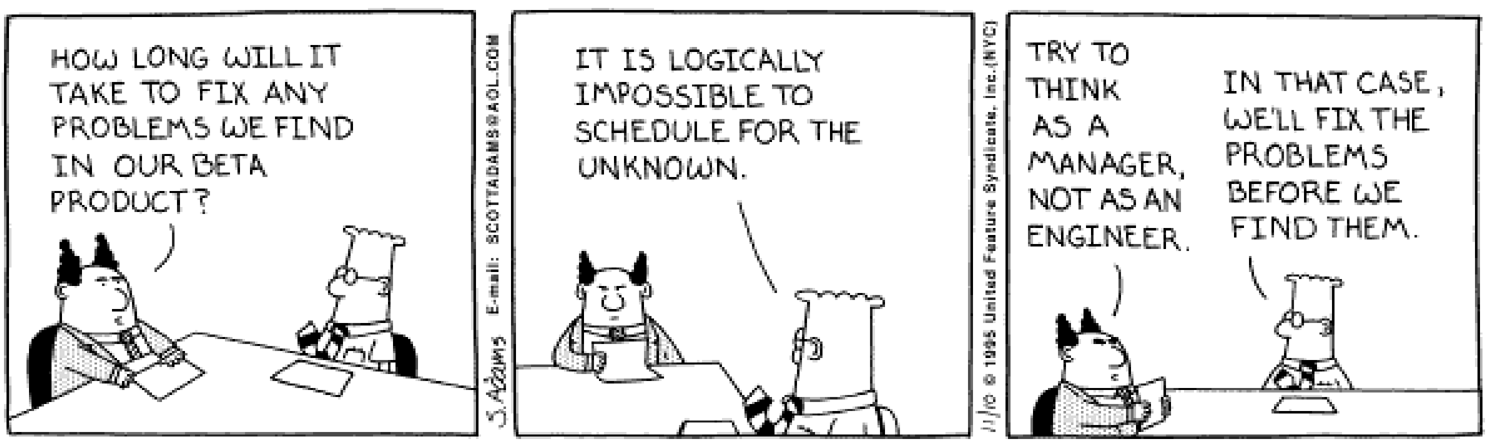
Issue arises at large & small level

At the level of developers, need way to manage this

- Both tools and procedures
  - We’ll be discussing CMT, a typical tool, but others exist
  - Individual collaborations have their own ways of sharing info

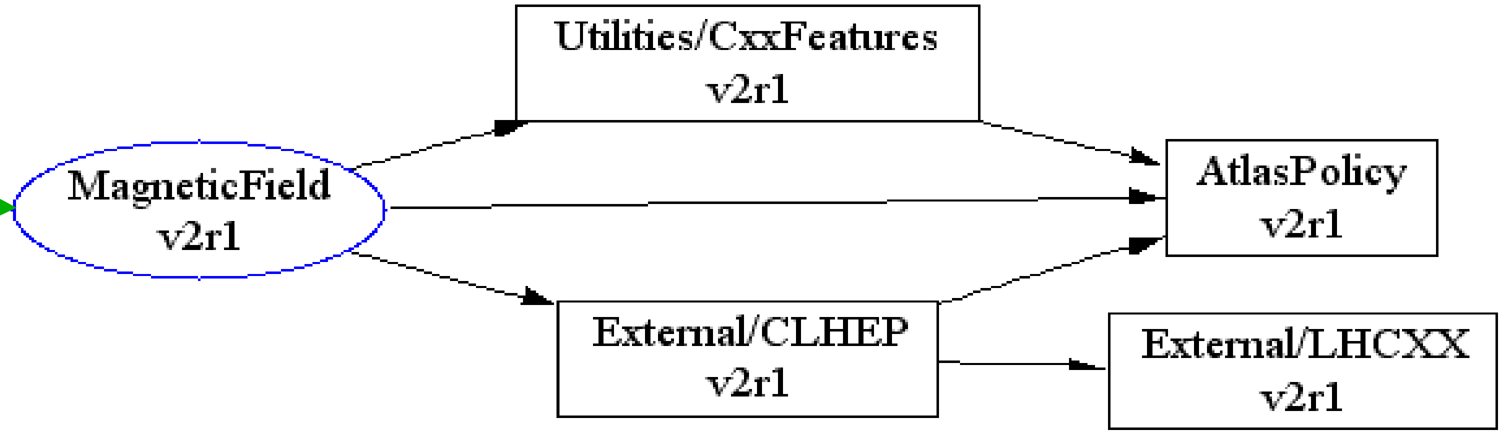
At the collaboration level, need procedures to ensure it works

- “Nightly builds”
  - Now common in HEP - Give early feedback on consistency problems
  - Many in industry moving toward “continuous integration”
- Not a complete solution by itself
  - Only works when people actually integrate early and often
- Reduces problems, but integration is still a lot of work



CMT: A modern tool example

Requirements file provides custom language for expressing our needs

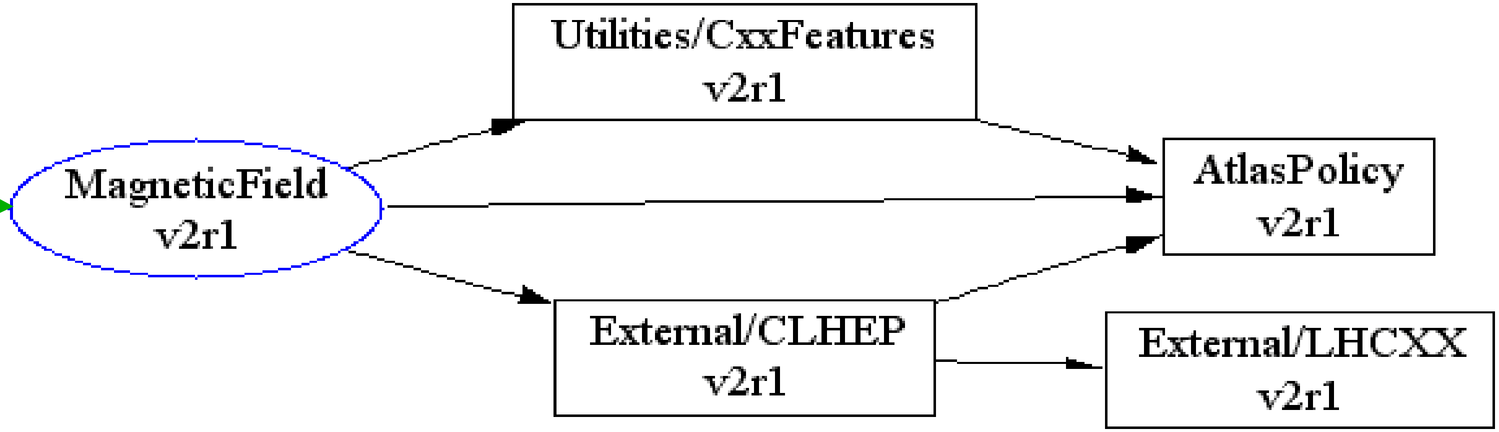



```
package MagneticField

author  Laurent Chevalier <laurent@hep.saclay.cea.fr>
author  Marc Virchaux <virchau@hep.saclay.cea.fr>

use AtlasPolicy v2r1
use CxxFeatures v2r1 Utilities
use CLHEP v2r1 External

include_dirs $(MAGNETICFIELDROOT)/MagneticField

branches MagneticField doc src test
...
```

Example from C. Arnault (LAL and Atlas)

CMT: A modern example

Requirements file provides custom language for expressing our needs



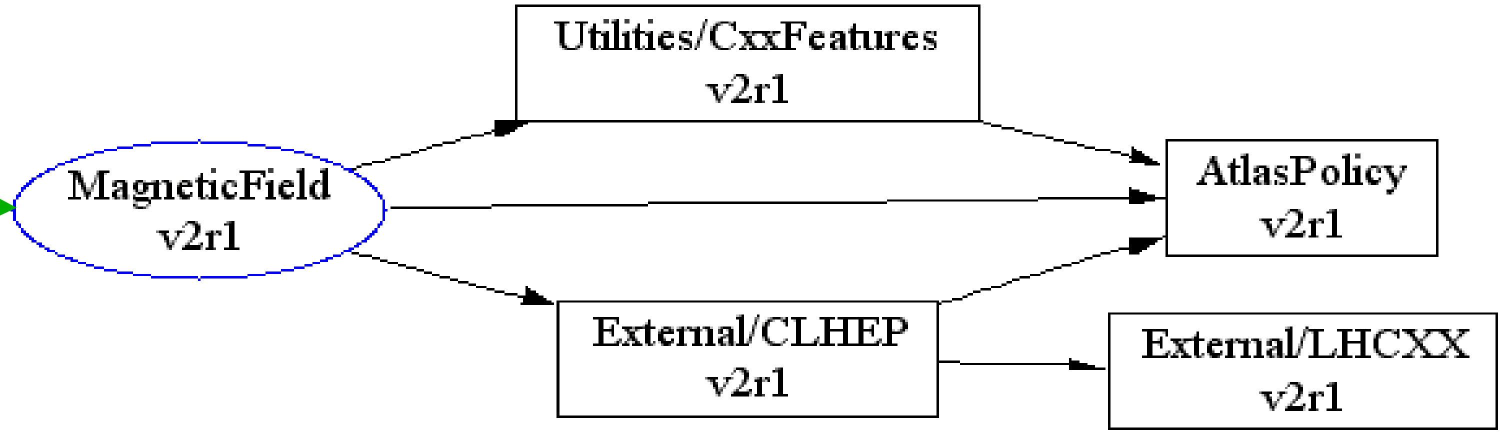

```
package MagneticField

author  Laurent Chevalier <laurent@hep.saclay.cea.fr>
author  Marc Virchaux <virchau@hep.saclay.cea.fr>

use AtlasPolicy v2r1
use CxxFeatures v2r1 Utilities
use CLHEP v2r1 External

include_dirs $(MAGNETICFIELDROOT)/MagneticField

branches MagneticField doc src test
...
```

Provides definitions for standard Atlas conventions (include paths, directory structure, default behavioural patterns, …)

CMT: A modern example

CERN School of Computing

Requirements file provides custom language for expressing our needs

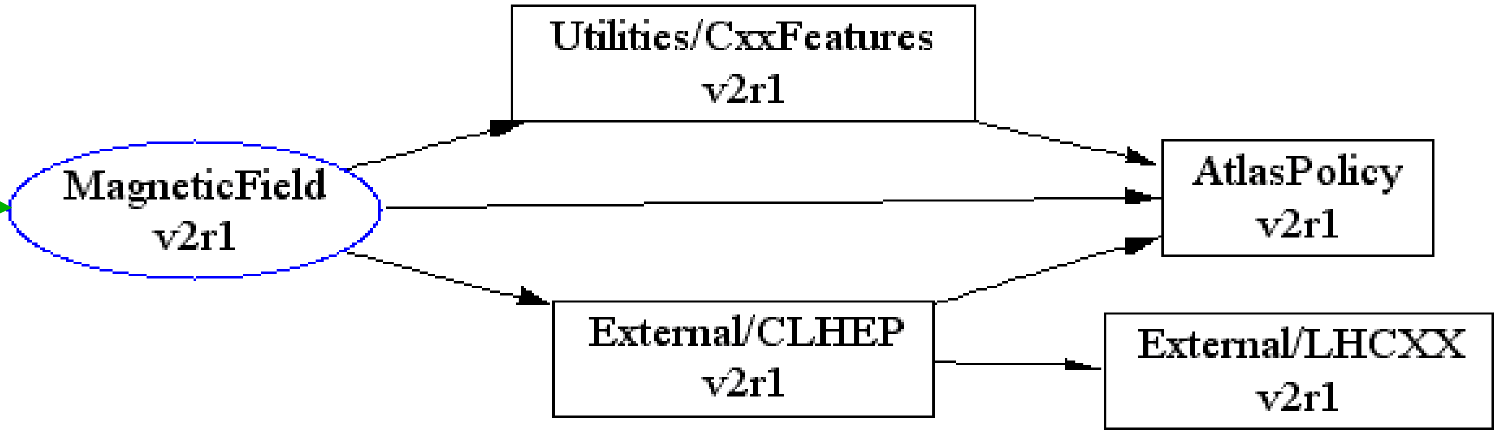



```
package MagneticField

author  Laurent Chevalier <laurent@hep.saclay.cea.fr>
author  Marc Virchaux <virchau@hep.saclay.cea.fr>

use AtlasPolicy v2r1
use CxxFeatures v2r1 Utilities
use CLHEP v2r1 External

include_dirs $(MAGNETICFIELDROOT)/MagneticField

branches MagneticField doc src test
...
```

An additional (non standard) include search path

CMT: A modern example

Requirements file provides custom language for expressing our needs




```
package MagneticField

author  Laurent Chevalier <laurent@hep.saclay.cea.fr>
author  Marc Virchaux <virchau@hep.saclay.cea.fr>

use AtlasPolicy v2r1
use CxxFeatures v2r1 Utilities
use CLHEP v2r1 External

include_dirs $(MAGNETICFIELDROOT)/MagneticField

branches MagneticField doc src test
...
```

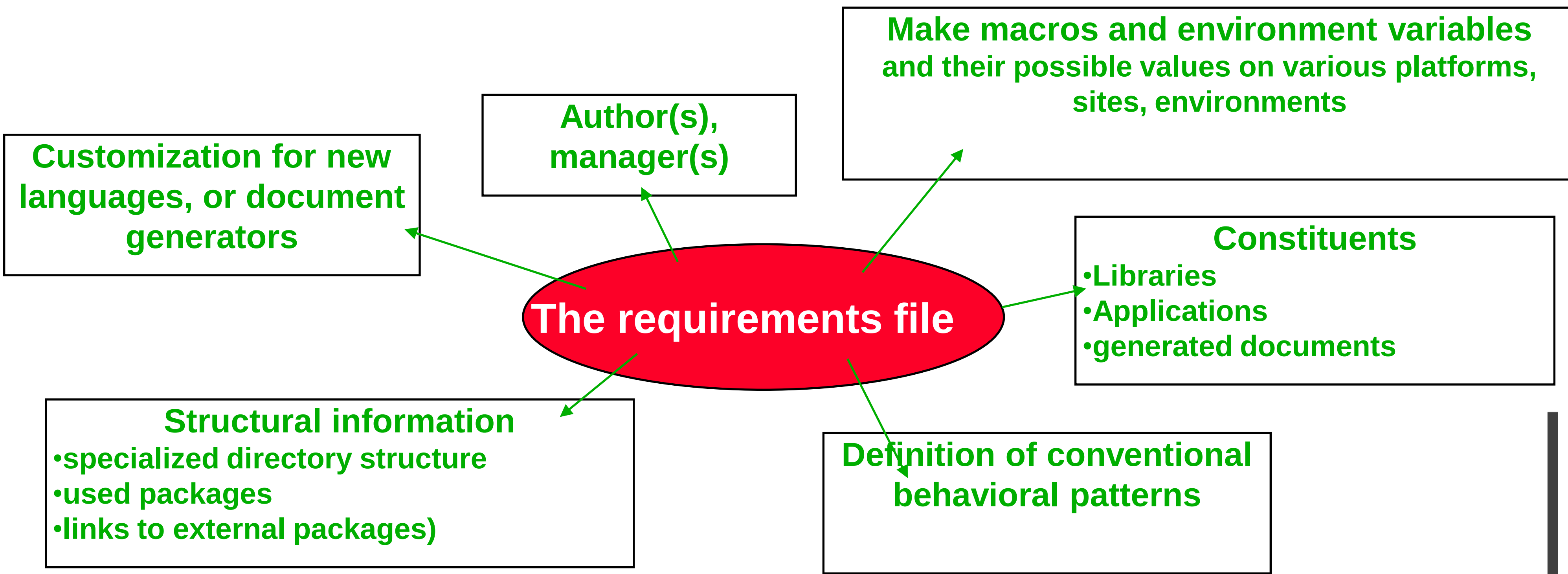
Describes additional subdirectories (branches) specific to this package

CERN School of Computing

CMT can reason from these

- **Find inconsistencies**
- **Create the include options needed for compile and link**
- **Connect to the correct prebuilt parts**

Includes more information that makes CMT more powerful for users:



Custom package structure: Describing a library

```
...
apply_pattern default_no_share_linkopts

library MagneticField -no_share \
  AbstractMagneticField.cxx \
  MagField.cxx \
  MagFieldFor.cxx \
  MagFieldGradient.cxx \
  Tableau.cxx \
  reamag.F \
  thanatos.F

...
```

Apply a “pattern” (defined in ATlasPolicy):

Provide client packages with information needed to link with static library provided this package.

Custom package structure: Describing a library

```
...
apply_pattern default_no_share_linkopts

library MagneticField -no_share \
  AbstractMagneticField.cxx \
  MagField.cxx \
  MagFieldFor.cxx \
  MagFieldGradient.cxx \
  Tableau.cxx \
  reamag.F \
  thanatos.F

...
```

This describes a (static) library and all its source files.

By default they are searched in ../src

The result will be

libMagneticField.a

Building a test program

```
...
application test -check ../test/main.cxx

private

macro data_file "/afs/cern.ch/atlas/offline/data/bmagatlas02.data"

macro test_pre_check "ln -s $(data_file) test.dat"
macro test_check_args "test.dat"
macro test_post_check "/bin/rm -f test.dat"

macro test_dependencies MagneticField
```

Create an application named test, with one source file

run with the command

> gmake check

Building a test program

```
...
application test -check ../test/main.cxx

private

macro data_file "/afs/cern.ch/atlas/offline/data/bmagatlas02.data”

macro test_pre_check "ln -s $(data_file) test.dat"
macro test_check_args "test.dat"
macro test_post_check "/bin/rm -f test.dat"

macro test_dependencies MagneticField
```

The following macro definitions are private to this package.

Client packages do not inherit these.

Building a test program

```
...
application test -check ../test/main.cxx

private

macro data_file "/afs/cern.ch/atlas/offline/data/bmagatlas02.data”

macro test_pre_check "ln -s $(data_file) test.dat"
macro test_check_args "test.dat"
macro test_post_check "/bin/rm -f test.dat"

macro test_dependencies MagneticField
```

Define data file to be used in the test procedure.

Building a test program

```
...
application test -check ../test/main.cxx

private

macro data_file "/afs/cern.ch/atlas/offline/data/bmagatlas02.data"

macro test_pre_check "ln -s $(data_file) test.dat"
macro test_check_args "test.dat"
macro test_post_check "/bin/rm -f test.dat"

macro test_dependencies MagneticField
```

These three standard make macros provide the parameters for the test procedure

Building a test program

```
...
application test -check ../test/main.cxx

private

macro data_file "/afs/cern.ch/atlas/offline/data/bmagatlas02.data"

macro test_pre_check "ln -s $(data_file) test.dat"
macro test_check_args "test.dat"
macro test_post_check "/bin/rm -f test.dat"

macro test_dependencies MagneticField
```

Assure that MagneticField target is always built before the test target.

This is useful when using the -j option of gmake

How do you know what's compatible?

Updated code might be fix, cause problems:

- Fix algorithmic bugs
- Add new capabilities
- Break interfaces
- Break assumptions

Collaborations enforce conventions via package versioning

- 'V01-02-03' as triplet of major, minor, patch numbers

'Bigger is better', but might break other things

Different major numbers mean they won't work together

A larger minor number is backward-compatible with a smaller one

Different patch numbers should work together

(But larger is still better)

CMT provides ways to ensure that requirements are met

Is that enough?

CERN School of Computing

When Boeing wanted to design the 747, they had two choices:

1. **Hire "SuperEngineer", who could do it alone**
2. **Hire 7,200 engineers and organize them to cooperate**

Which did they choose?

Why?

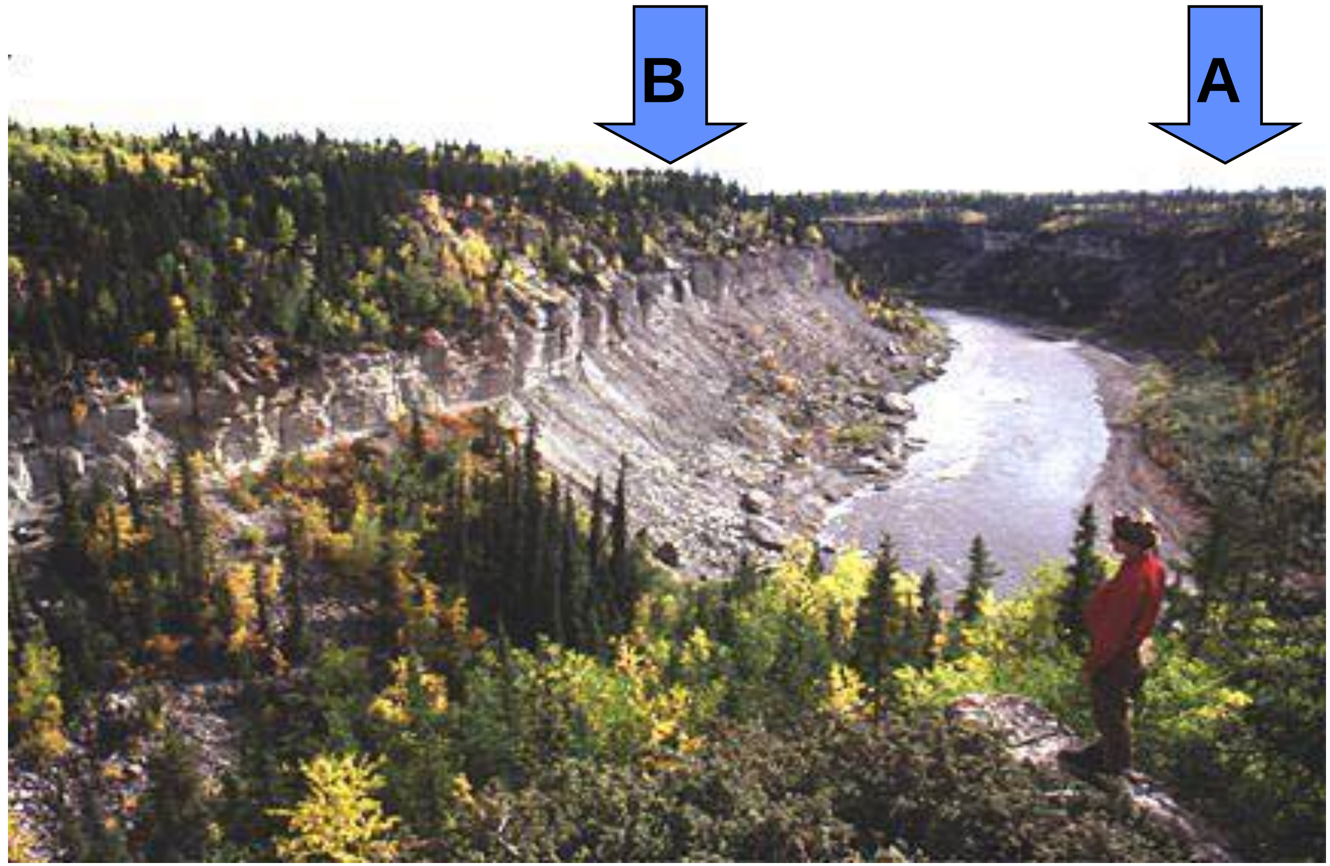
What can we learn from this?

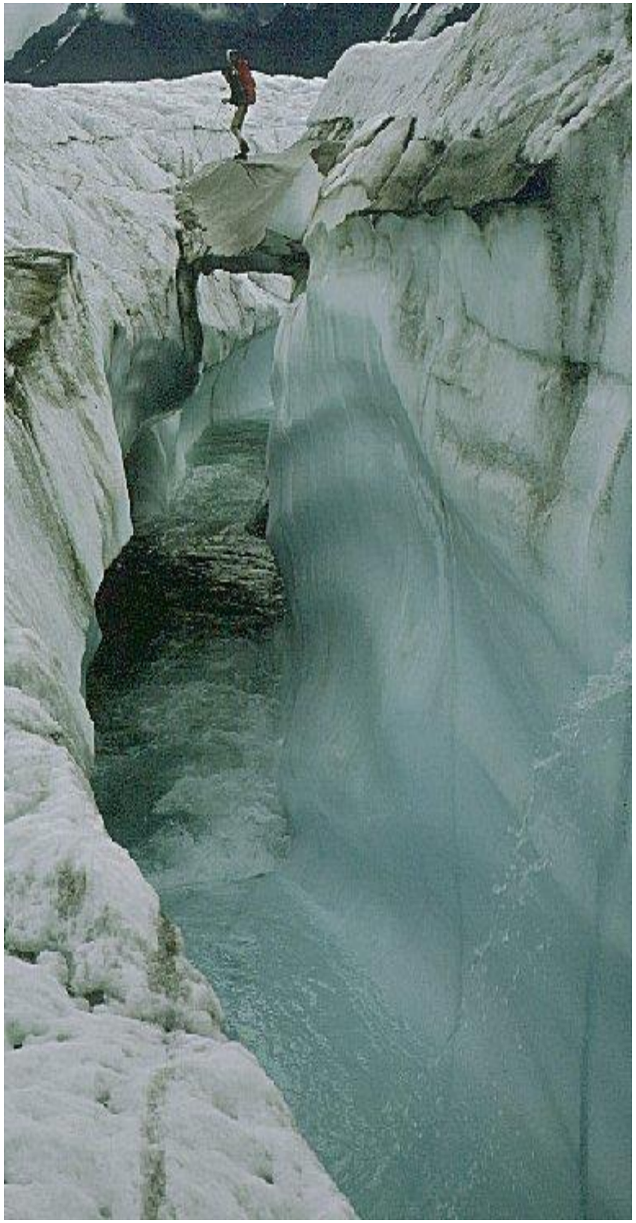
CERN School of Computing

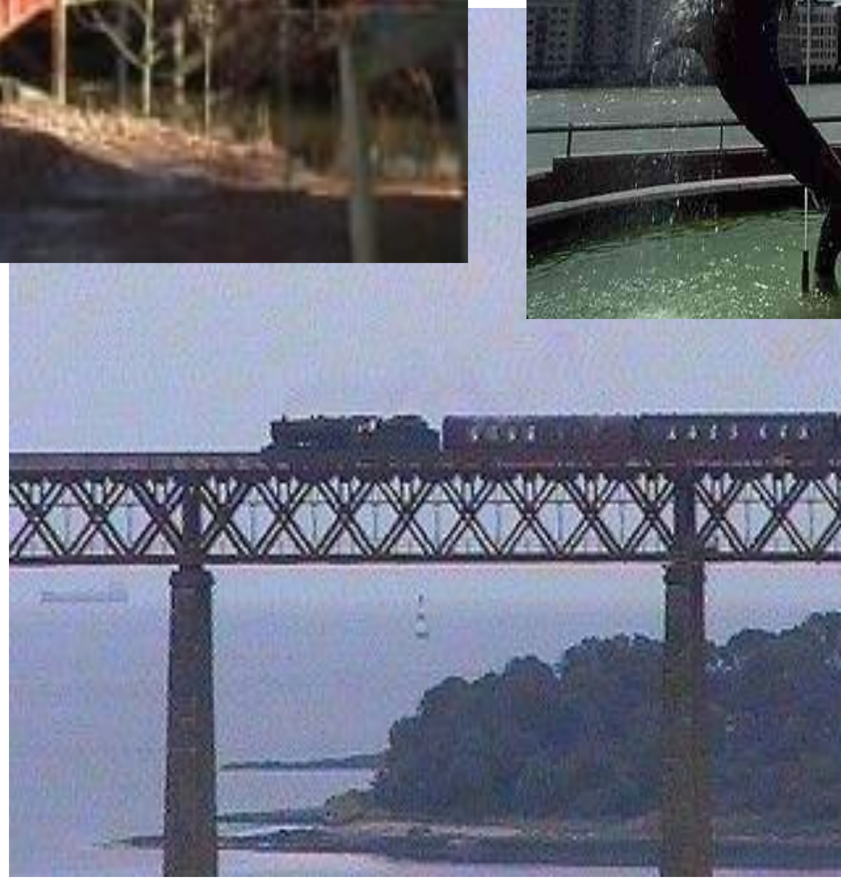
This is where iterative development comes in…

Imagine the project is not to build software but a bridge…

Initial Requirements: A to B







Successful Development Program!

Analogy shows successful iterations:

- **The basic *product* existed from the first iteration and met the primary requirement: Connect A to B**
- **Early emphasis on defining the architecture**
- **Basic architecture remained the same over iterations**
- **Extra functionality/reliability/robustness was added at each iteration**
- **Each iteration required more analysis, design, implementation and testing**
- **Use case (requirements) driven**
  - Does what the users want - not what the developers think is cool

Some limits to analogy:

It took people centuries to figure out how to build big bridges

And we developed engineering processes to do the big ones!

Little of the early cycles survived in final one

How to pick what goes in the next iteration?

Choice of additions for an iteration is *risk driven*

- Early development focuses on parts with the highest risk and uncertainty

  Avoids investing resources in a project that is not feasible

- But it has to do something basically useful

  So all involved will take it seriously

Similar issues during deployment

“We need to get Z working”

“We’ve just found the problem with Y”

“X was just badly broken!”

“The conference is in two months, and W keeps changing!”

What can go wrong?

Advantages of Iterative and Incremental Development

Complexity is never overwhelming
- Only tackle small bits at a time
- Avoid *analysis paralysis* and *design decline*

Continuous feedback from users
- Provides input to the direction of subsequent iterations

Developers skills can *grow* with the project
- Don't need to apply latest techniques/technology at the start
- Get used to delivering finished software

Requirements can be modified
- Each iteration is a mini-project (analysis, design….)

Note that these benefits come from completing, deploying and using the iterations!

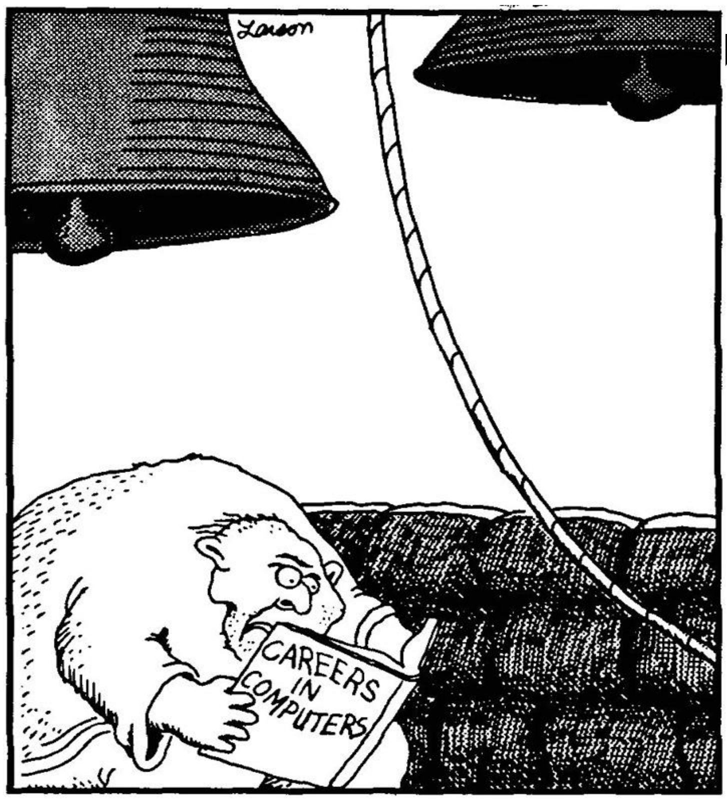
Lecture summary

Software engineering is the art of building complex computer systems

It's ideas and techniques spring from our need to handle size & complexity

As you do your own work & develop your own skills, consider:

- **How your effort effects or contributes to things 10X, 100X, 1000X larger**
- **s different/better when it's your problem**



Exercises

1) Demonstration of a test framework
2) Practice debugging using a test framework
3) Demonstration of a profiling tool
4) Practice tuning small applications

If you want experience with CVS, we've got optional exercises:

A) Simple use of CVS B) More advanced CVS, showing how conflicts are handled

If you want some more practice with performance tuning, we've got two optional exercises:

5) Understanding, updating and tuning a larger application
6) Tuning a sample RSA encryption/decryption application

7) Simple release activities with CMT
8) Releasing code changes with CMT
9) Managing configuration conflicts
10) Project - Joint Development

Instruction sheets are available via web browser at

file:/home/jake/CSC/index.html