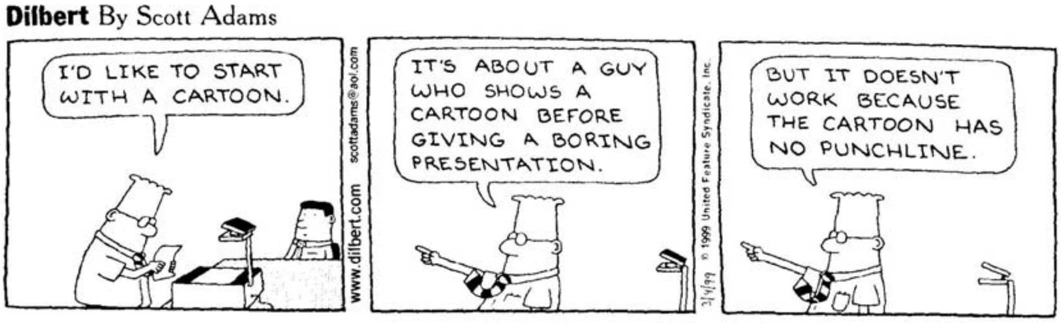
CERN School of Computing

Large Projects & Software Engineering


Dilbert By Scott Adams
I'D LIKE TO START WITH A CARTOON.
scottadams@aol.com
www.dilbert.com
IT'S ABOUT A GUY WHO SHOWS A CARTOON BEFORE GIVING A BORING PRESENTATION.
3/1/99 © 1999 United Feature Syndicate, Inc.
BUT IT DOESN'T WORK BECAUSE THE CARTOON HAS NO PUNCHLINE.

With thanks to Bob Jones for ideas and illustrations

CERN School of Computing

Why spend so much time talking about "Software Process"?

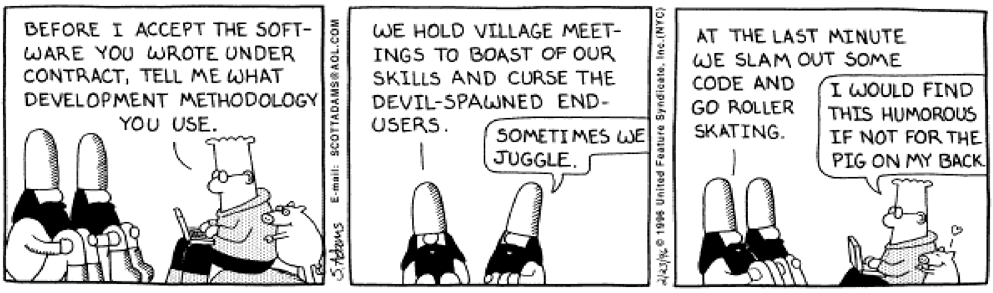
How do you create software?

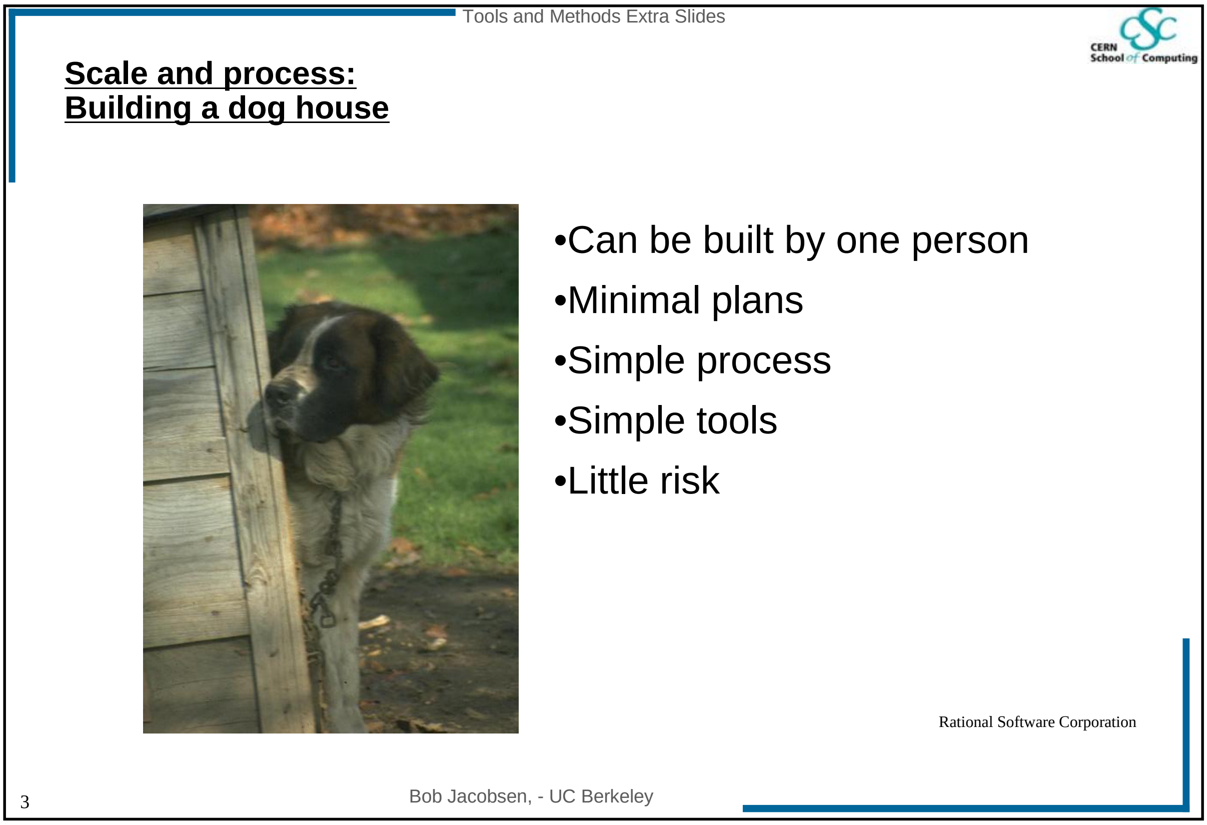
- **Lots of parts: Writing, documenting, testing, sharing, fixing, ….**
- **Usually done by lots of people**

"Process" is just a big word for how they do this

- **Exists whether you talk about it or not**

"Why do we have to formalize this?"

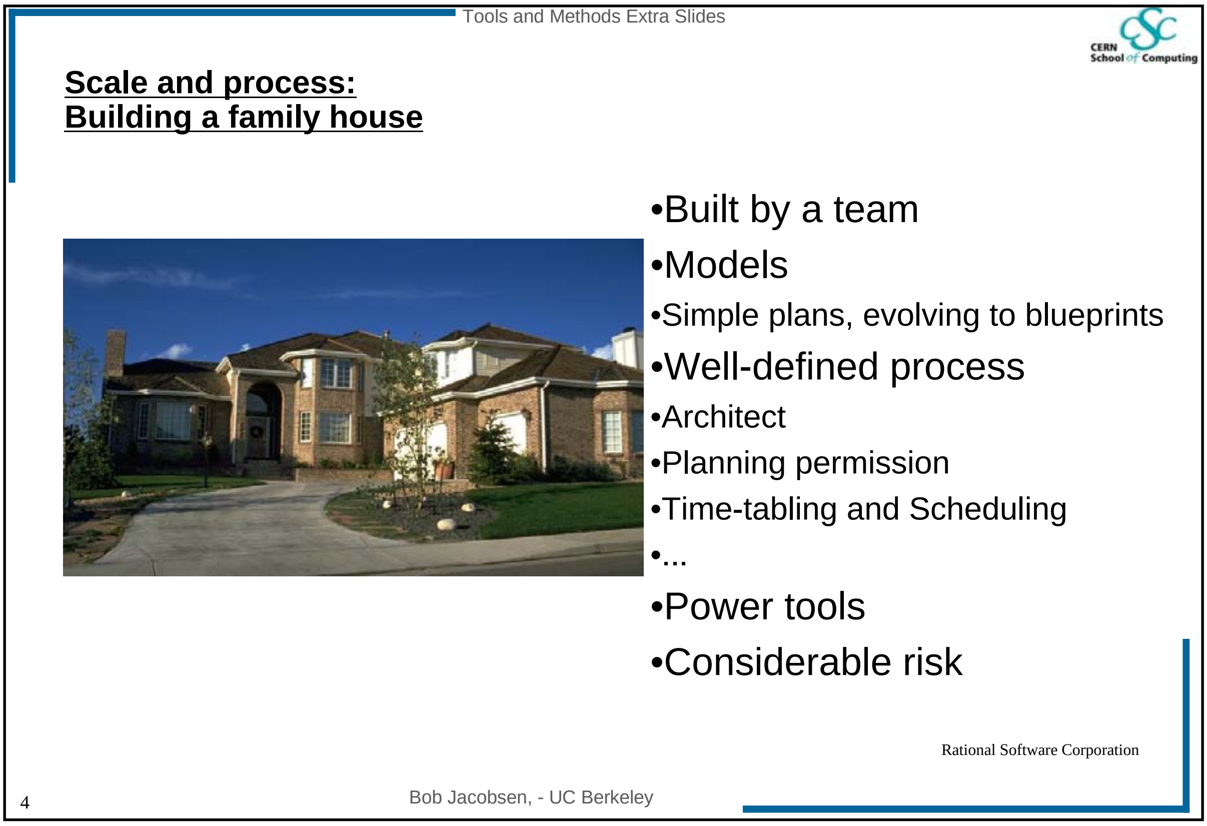

BEFORE I ACCEPT THE SOFTWARE YOU WROTE UNDER CONTRACT, TELL ME WHAT DEVELOPMENT METHODOLOGY YOU USE.
E-mail: SCOTTADAMS@AOL.COM
S. Adams
WE HOLD VILLAGE MEETINGS TO BOAST OF OUR SKILLS AND CURSE THE DEVIL-SPAWNED END-USERS.
SOMETIMES WE JUGGLE.
2/13/96 © 1996 United Feature Syndicate, Inc.(NYC)
AT THE LAST MINUTE WE SLAM OUT SOME CODE AND GO ROLLER SKATING.
I WOULD FIND THIS HUMOROUS IF NOT FOR THE PIG ON MY BACK.



Scale and process: Building a dog house

- Can be built by one person
- Minimal plans
- Simple process
- Simple tools
- Little risk

Rational Software Corporation



Scale and process: Building a family house

- Built by a team
- Models
- Simple plans, evolving to blueprints
- Well-defined process
- Architect
- Planning permission
- Time-tabling and Scheduling
- ...
- Power tools
- Considerable risk

Rational Software Corporation

Scale and process: Building a skyscraper

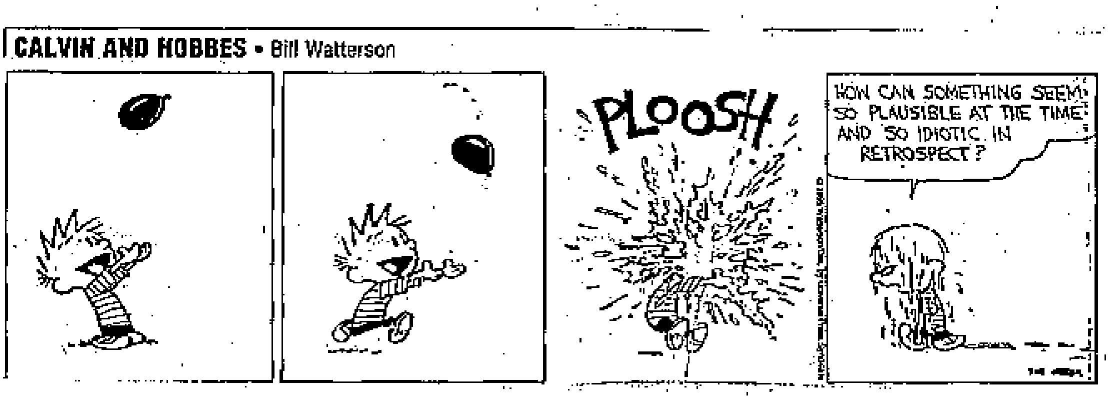
- Built by many companies
- Modeling
  - Simple plans, evolving to blueprints
  - Scale models
  - Engineering plans
- Well-defined process
  - Architectural team
  - Political planning
  - Infrastructure planning
  - Time-tabling and scheduling
  - Selling space
- Heavy equipment
- Major risks

Rational Software Corporation

Why do software projects fail?

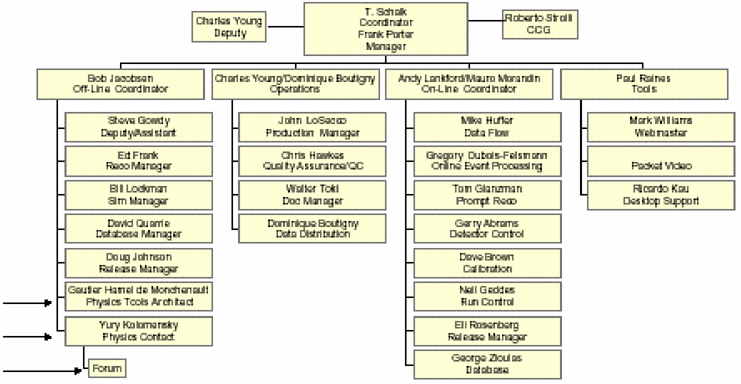
Even if you do produce the code it does not guarantee that the project will be a success

There are many other factors (both internal and external) that can affect the success of a project...


CALVIN AND HOBBES • Bill Watterson
PLOOSH
HOW CAN SOMETHING SEEM SO PLAUSIBLE AT THE TIME AND SO IDIOTIC IN RETROSPECT?

Communication explosion

More people means *more* time communicating which means more misunderstandings and *less* time for the software



Charles Young
Deputy

T. Schalk
Coordinator
Frank Porter
Manager

Roberto Stroili
CCG

Bob Jacobsen
Off-Line Coordinator

- Steve Gowdy
  Deputy/Assistant
- Ed Frank
  Reco Manager
- Bill Lockman
  Sim Manager
- David Quarrie
  Database Manager
- Doug Johnson
  Release Manager
- Gautier Hamel de Monchenault
  Physics Tools Architect
- Yury Kolomensky
  Physics Contact
- Forum

Charles Young/Dominique Boutigny
Operations

- John LoSecco
  Production Manager
- Chris Hawkes
  Quality Assurance/QC
- Walter Toki
  Doc Manager
- Dominique Boutigny
  Data Distribution

Andy Lankford/Mauro Morandin
On-Line Coordinator

- Mike Huffer
  Data Flow
- Gregory Dubois-Felsmann
  Online Event Processing
- Tom Glanzman
  Prompt Reco
- Gerry Abrams
  Detector Control
- Dave Brown
  Calibration
- Neil Geddes
  Run Control
- Eli Rosenberg
  Release Manager
- George Zioulas
  Database

Paul Raines
Tools

- Mark Williams
  Webmaster
- Packet Video
- Ricardo Kau
  Desktop Support

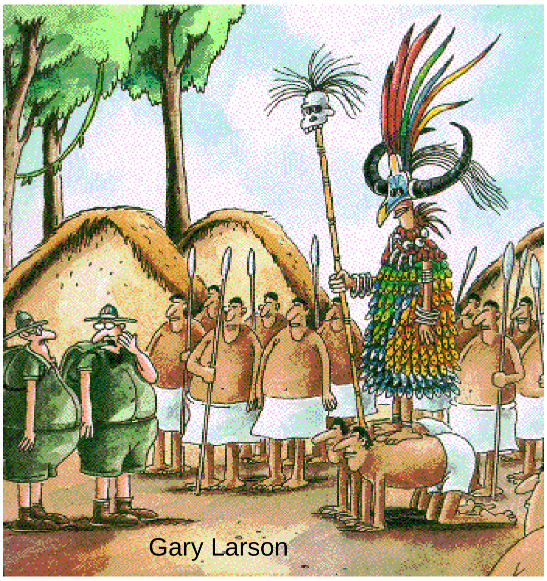
I JUST HEARD THAT ALL OUR TOP EXECUTIVES GOT LOCKED IN A CONFERENCE ROOM AND STARVED TO DEATH.
www.dilbert.com
S. Adams
WHY DIDN'T THEY USE THE PHONE TO CALL FOR HELP?
4/11/98 © 1998 United Feature Syndicate, Inc.
ONE WEEK AGO...
IT'S AGREED: WE DIAL 8.3 TO GET AN OUTSIDE LINE.
UH-OH. THIS ONE DOESN'T DO DECIMALS.

Why software projects fail...

Undefined responsibilities

"Hey... this could be the chief"

Too little responsibility can cause a lot of confusion & embarrassing mistakes

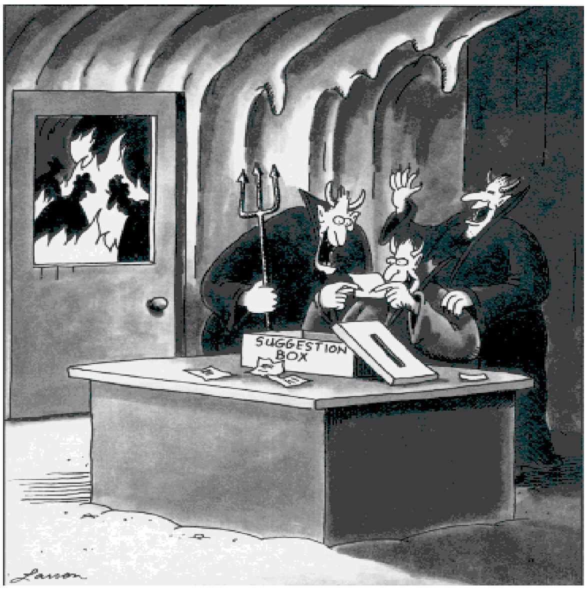


Gary Larson

Why software projects fail...

CERN School of Computing

Missed user requirements



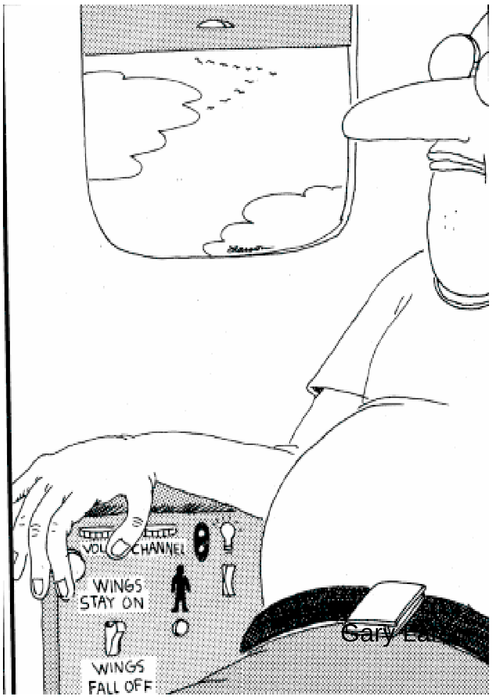
We're not smart enough to know everything people want the system to do; we need to ask!

Why software projects fail...

CERN School of Computing

Badly defined interfaces

Fumbling for his recline button, Bob unwittingly instigates a disaster



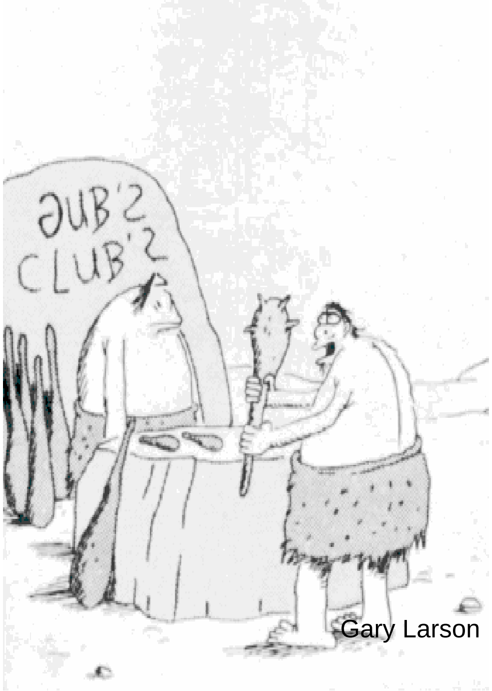
Spend the time to design and test good interfaces

Why software projects fail...

Creeping featurism

“No, no… Not this one. Too many bells and whistles”

Focus on what the users are asking for, not what the developers think is cool



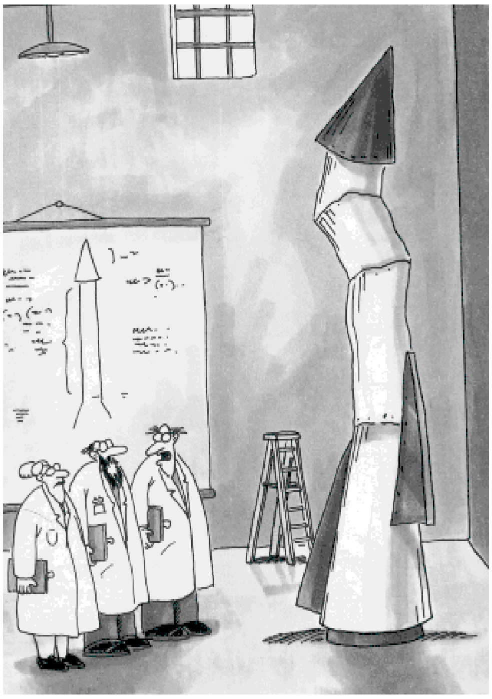
Gary Larson

Why software projects fail...

Unrealistic goals

“It’s time we face reality, my friends… We’re not exactly rocket scientists”

Analysis and design would make it clear the project is not feasible



R. Brun

The life time of HEP software

Software is a long-term commitment

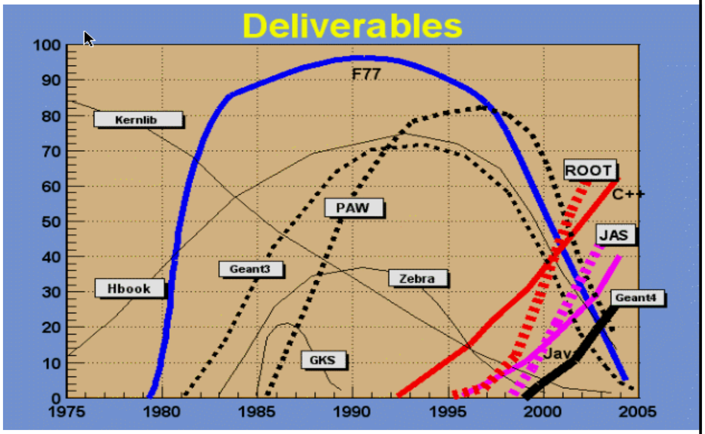
Users like stable and maintained systems

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 !!



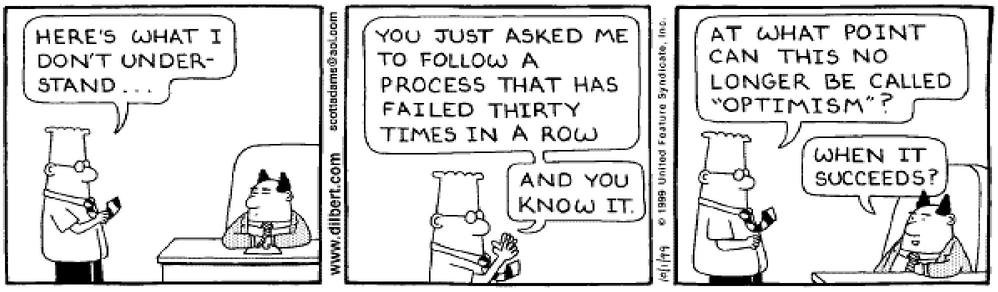
Many releases of the software are needed over its lifetime to fix bugs, add new features, support new platforms etc

How do we cope?

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:


HERE'S WHAT I DON'T UNDERSTAND . . .
scottadams@aol.com
www.dilbert.com
YOU JUST ASKED ME TO FOLLOW A PROCESS THAT HAS FAILED THIRTY TIMES IN A ROW
AND YOU KNOW IT.
© 1999 United Feature Syndicate, Inc.
AT WHAT POINT CAN THIS NO LONGER BE CALLED "OPTIMISM"?
WHEN IT SUCCEEDS?

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!

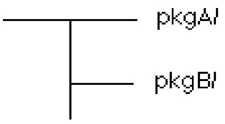
- Works great for small projects
- But runs into several levels of scaling problems

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 parts
- But software has cross-connections
- Need structure that works for both

And still need to be able to build the code

Handling complicated builds

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

BaBar (offline production code only):

- 350 packages
- 14,000 files
- 6 million lines of source

Some of these are large “for historical reasons”

But that’s true of just about any project

CVS checkout: 41 minutes

Build from scratch: 14 hours

Spread across multiple production machines; never did complete on laptop

“gmake” with one change: about 6 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

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

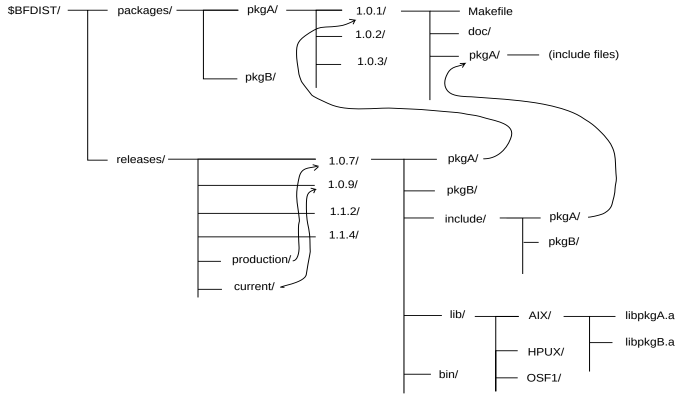
“Architecture”: A particular type of computer

hardware, software, even location

Simple Example: SRT (SoftRelTools)

- 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



Typical use:

- Create an area for your own work
  - Specify the production release you want as context
- Do a CVS checkout of 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 into CVS
    - Now they're safe, and colleagues can get changes
  - Tag CVS
    - So you can tell your colleagues how to get these
  - 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

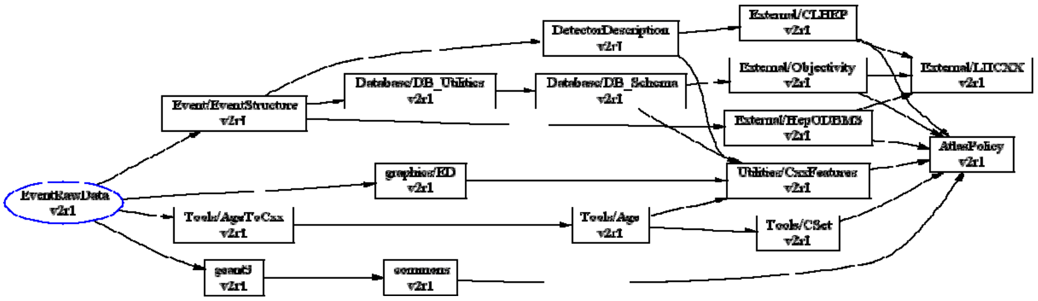
Help distributing the workload

- 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”



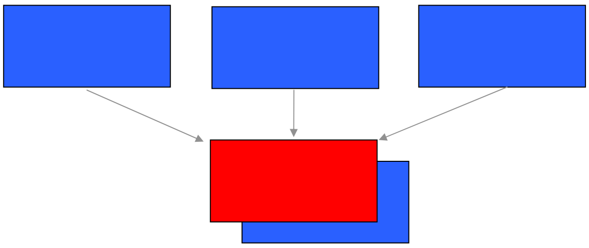
Software strongly depends on other software

- 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



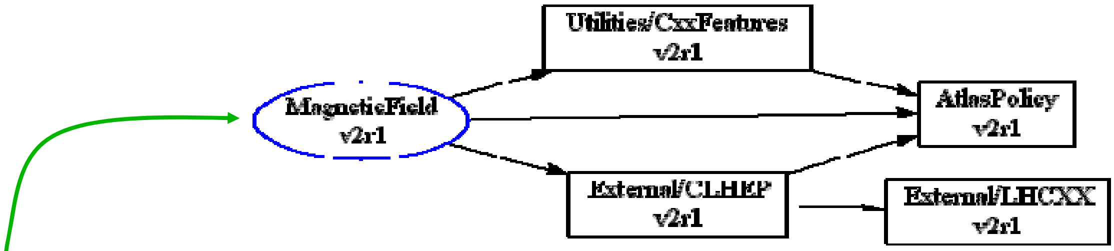
One solution: Separate stable interfaces from evolving implementations

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

CMT: A modern example

Requirements file provides custom language for expressing our needs

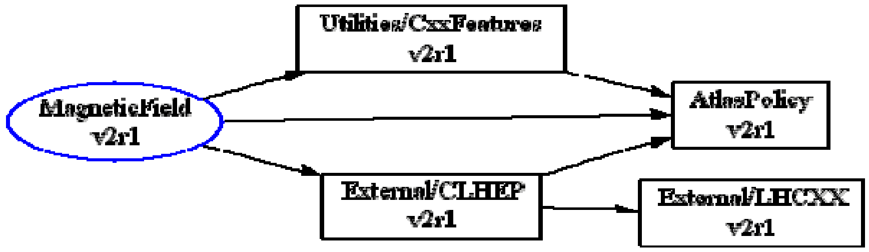


```
package MagneticField

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
...
```

Example from C. Arnault (LAL and Atlas)

CMT: A modern example

Requirements file provides custom language for expressing our needs

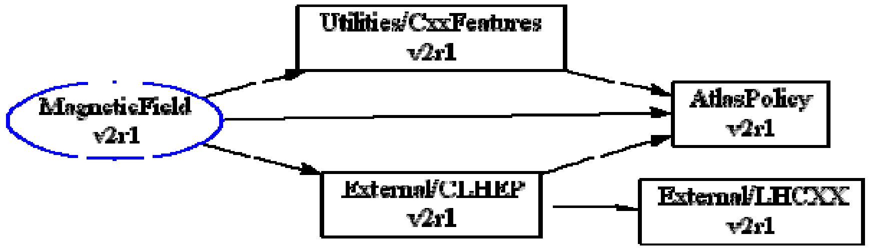


```
package MagneticField

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, ...)

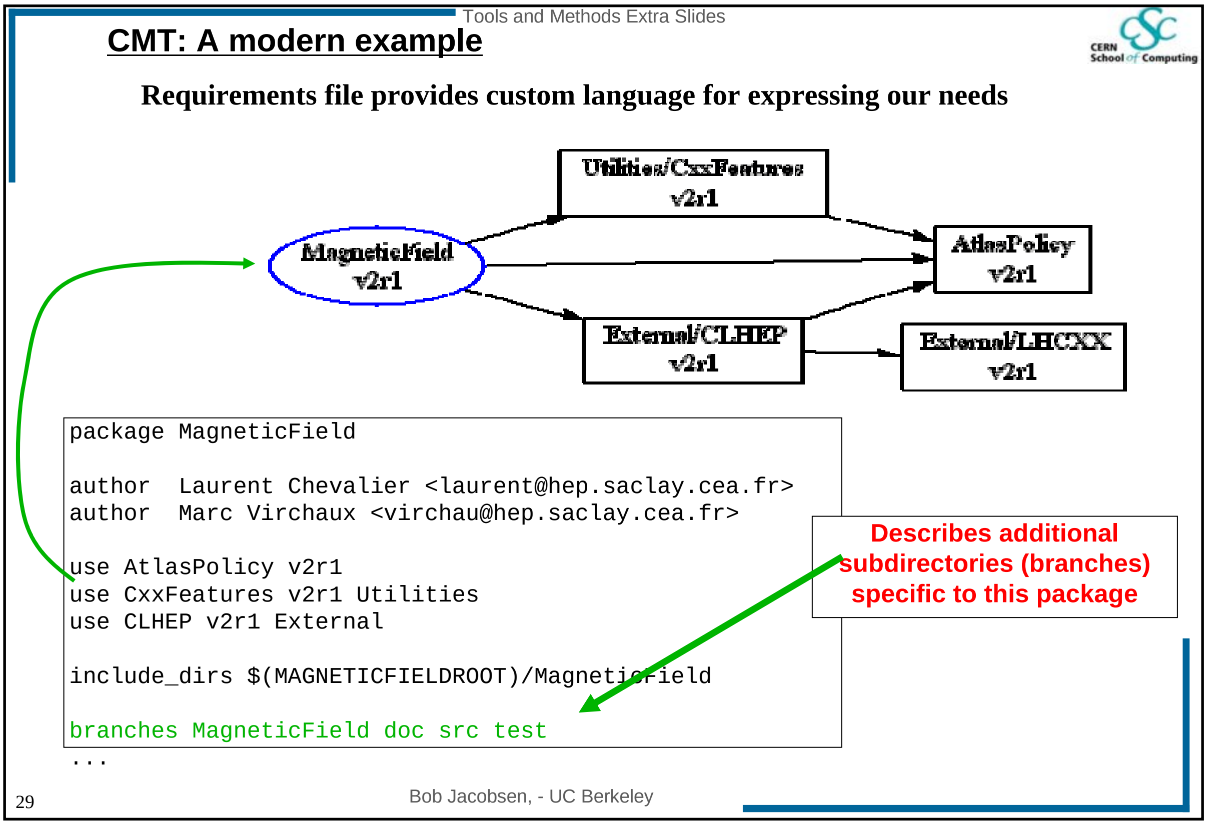
CMT: A modern example

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

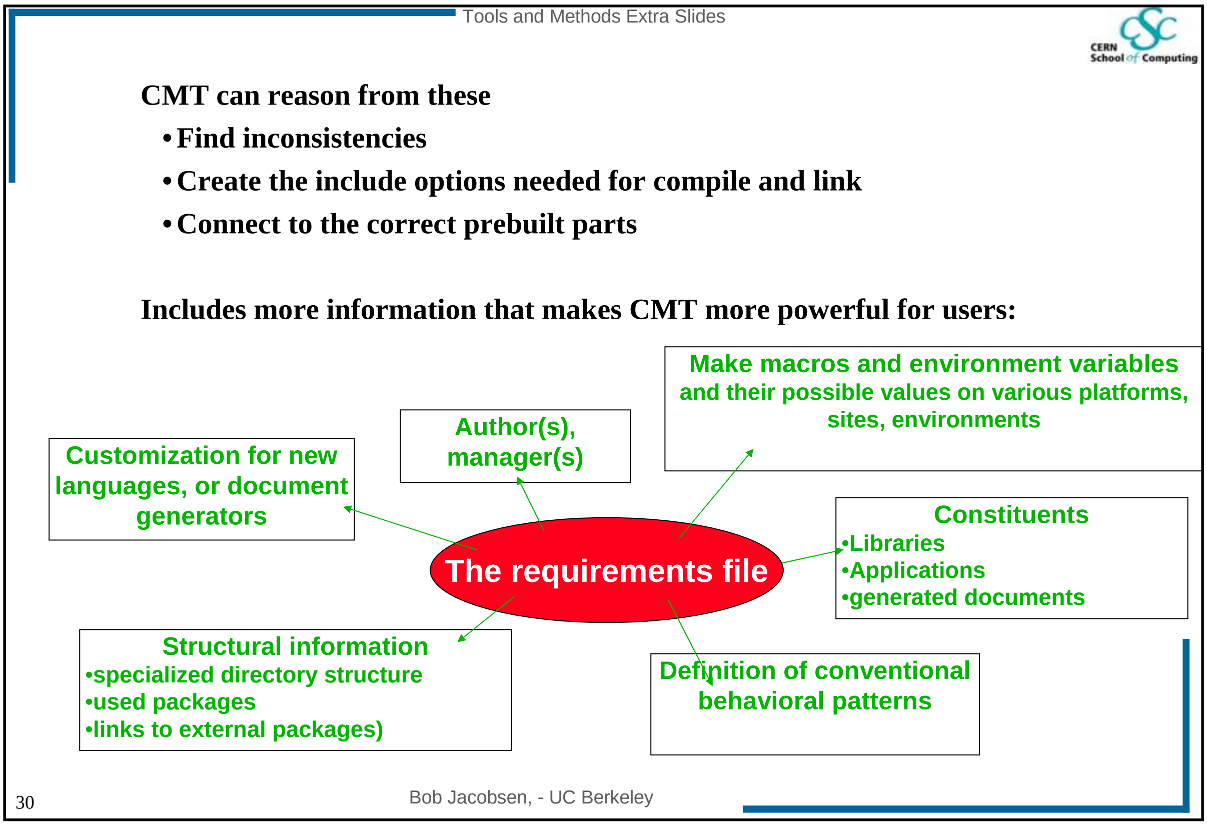


CMT: A modern example

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



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:

Make macros and environment variables and their possible values on various platforms, sites, environments

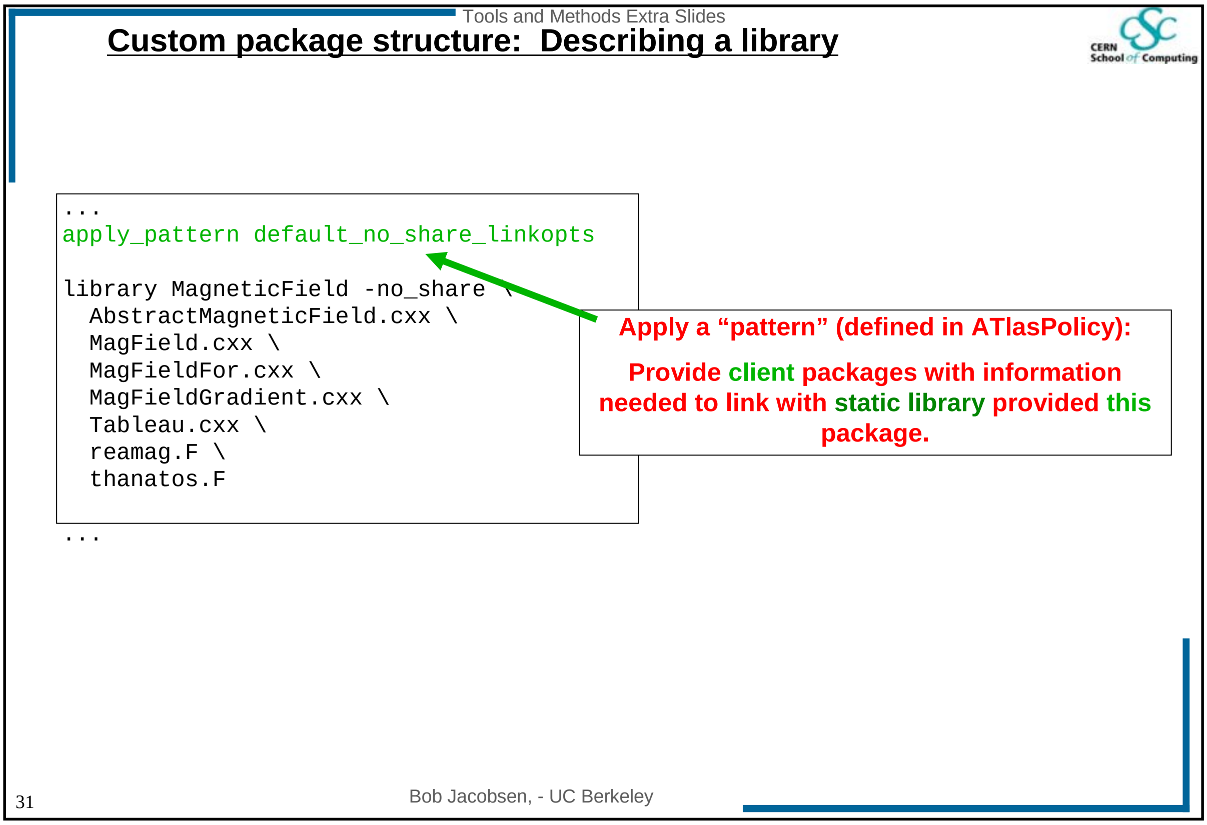
Author(s), manager(s)

Customization for new languages, or document generators

Constituents
•Libraries
•Applications
•generated documents

Structural information
•specialized directory structure
•used packages
•links to external packages)

Definition of conventional behavioral patterns

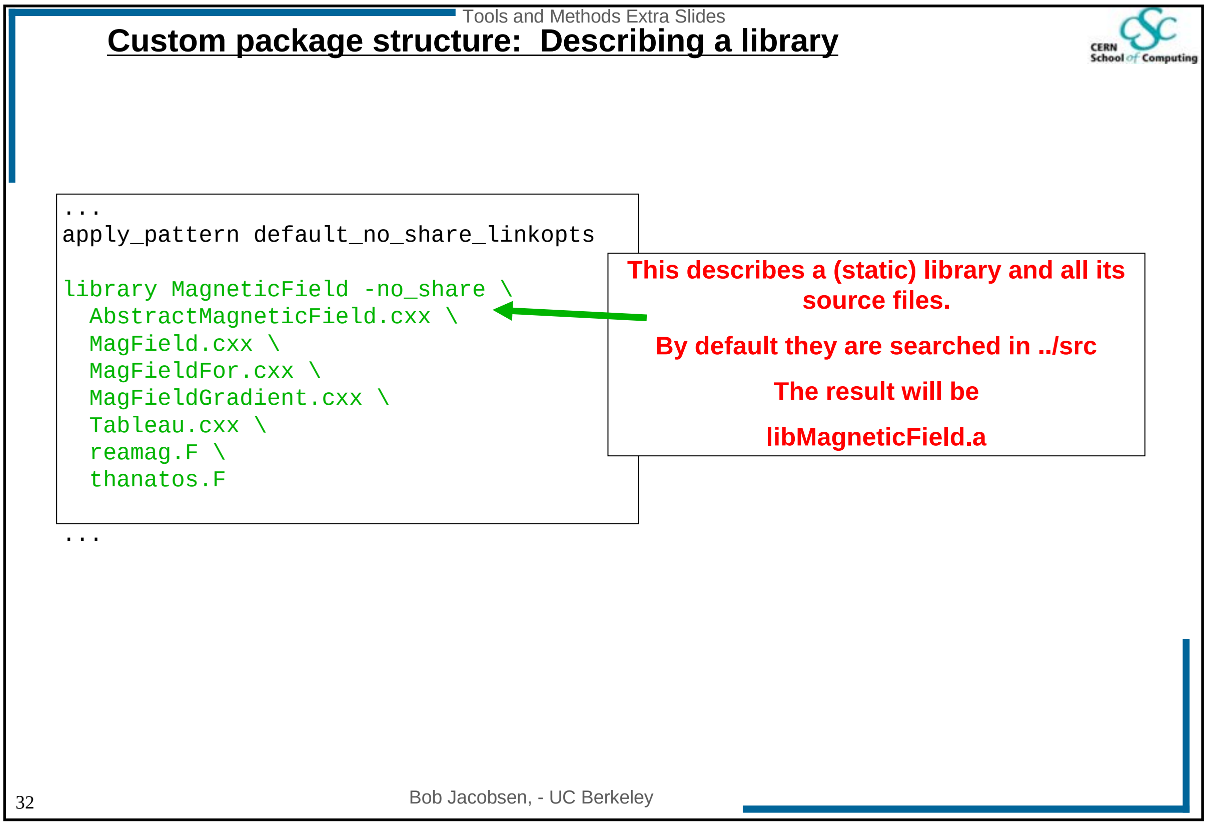


Custom package structure: Describing a library

```
...
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

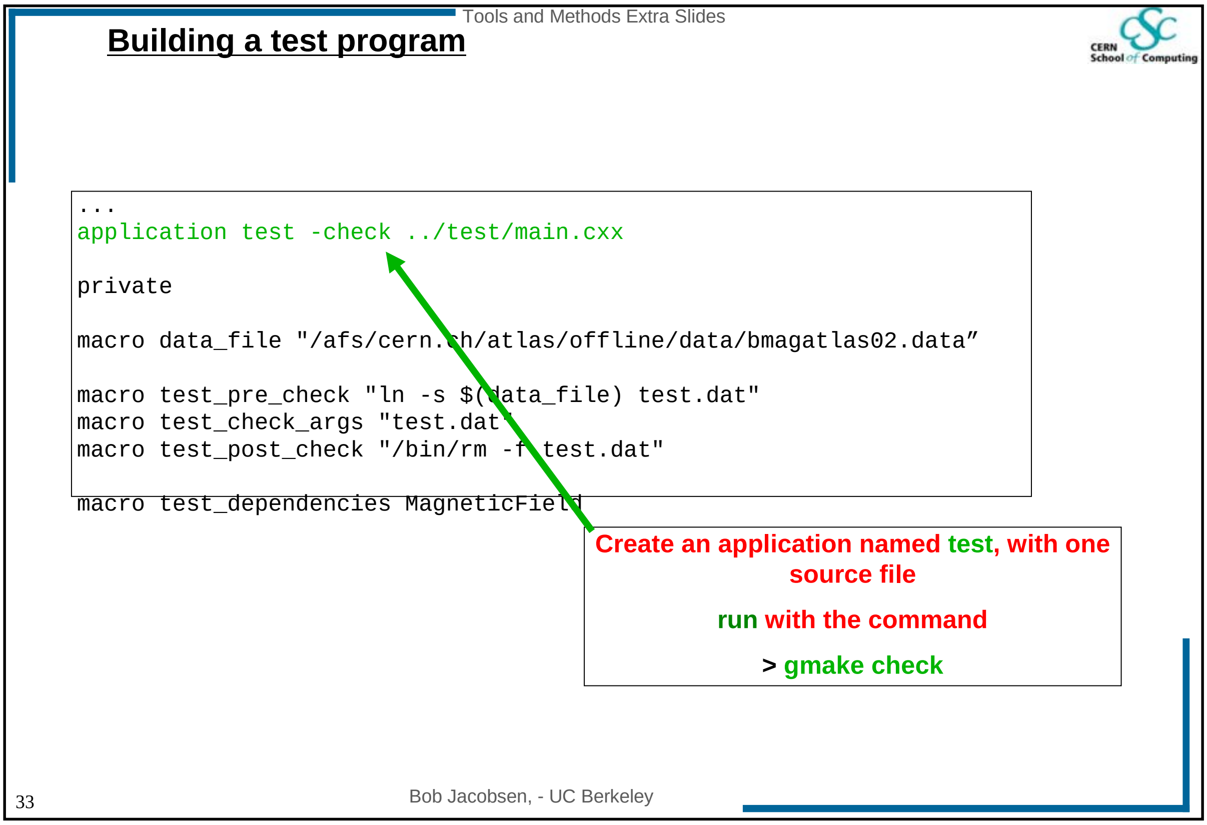
```
...
apply_pattern default_no_share_linkopts

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



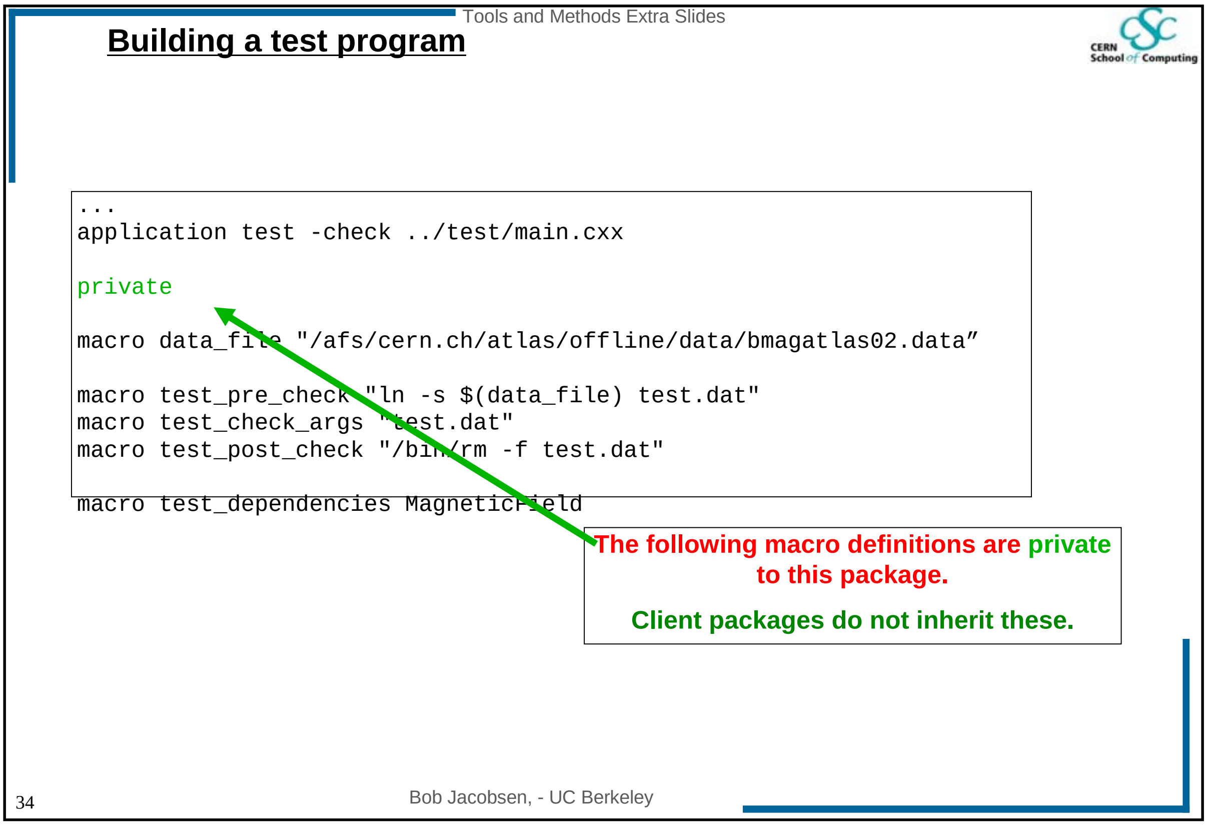
Building a test program

```
...
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

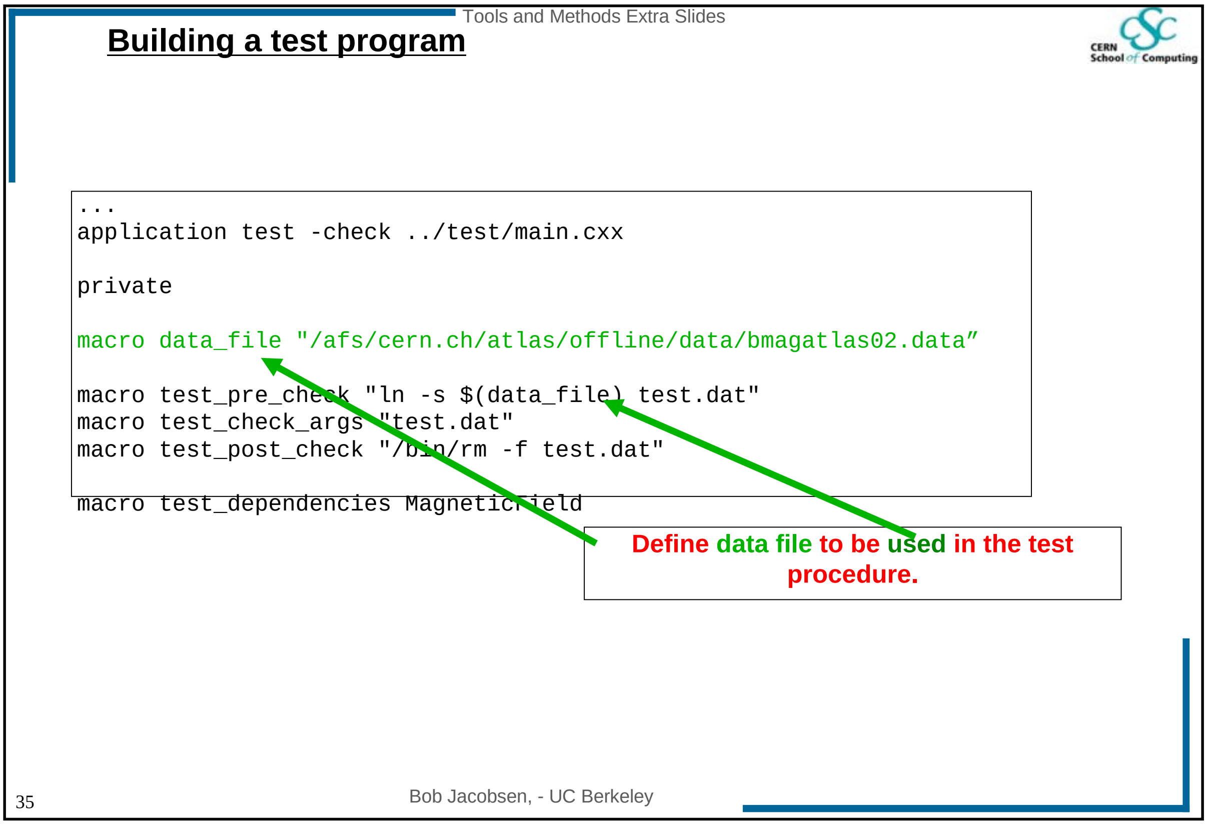


Building a test program

```
...
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.

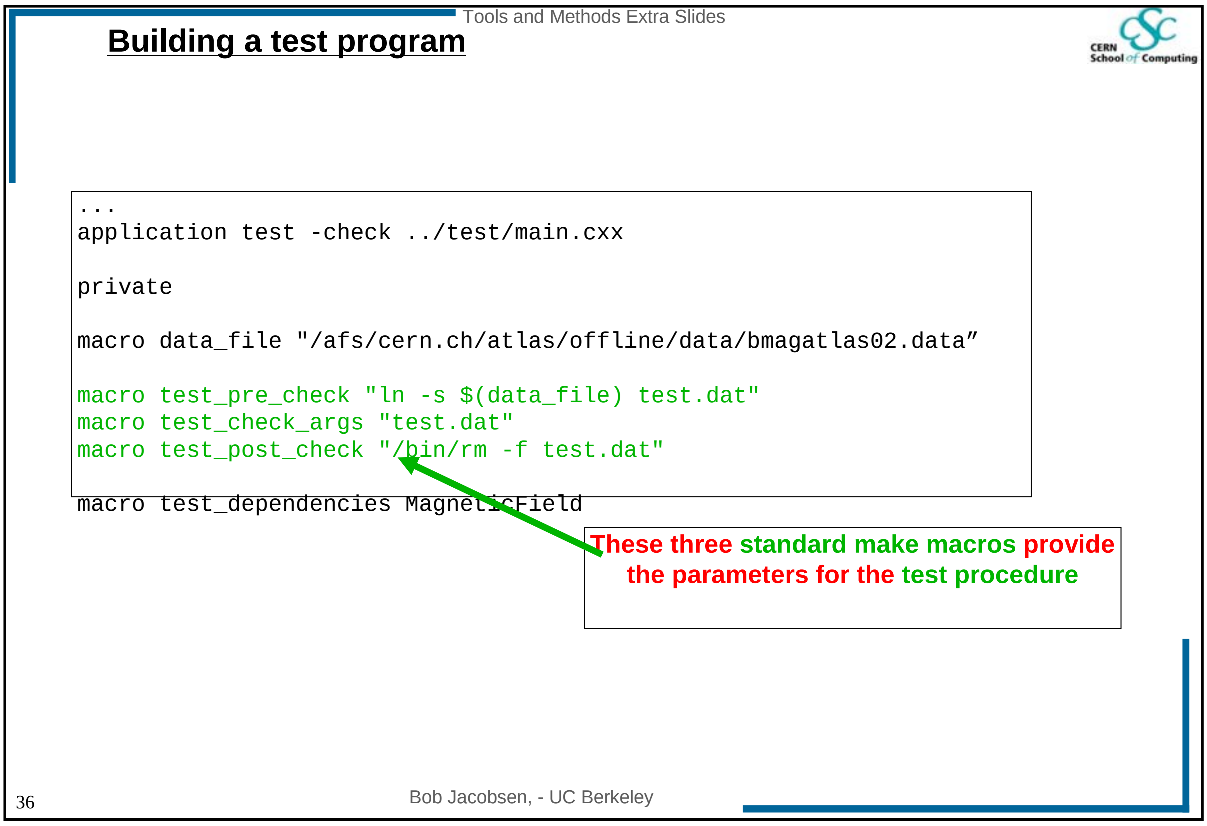
Client packages do not inherit these.



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
```

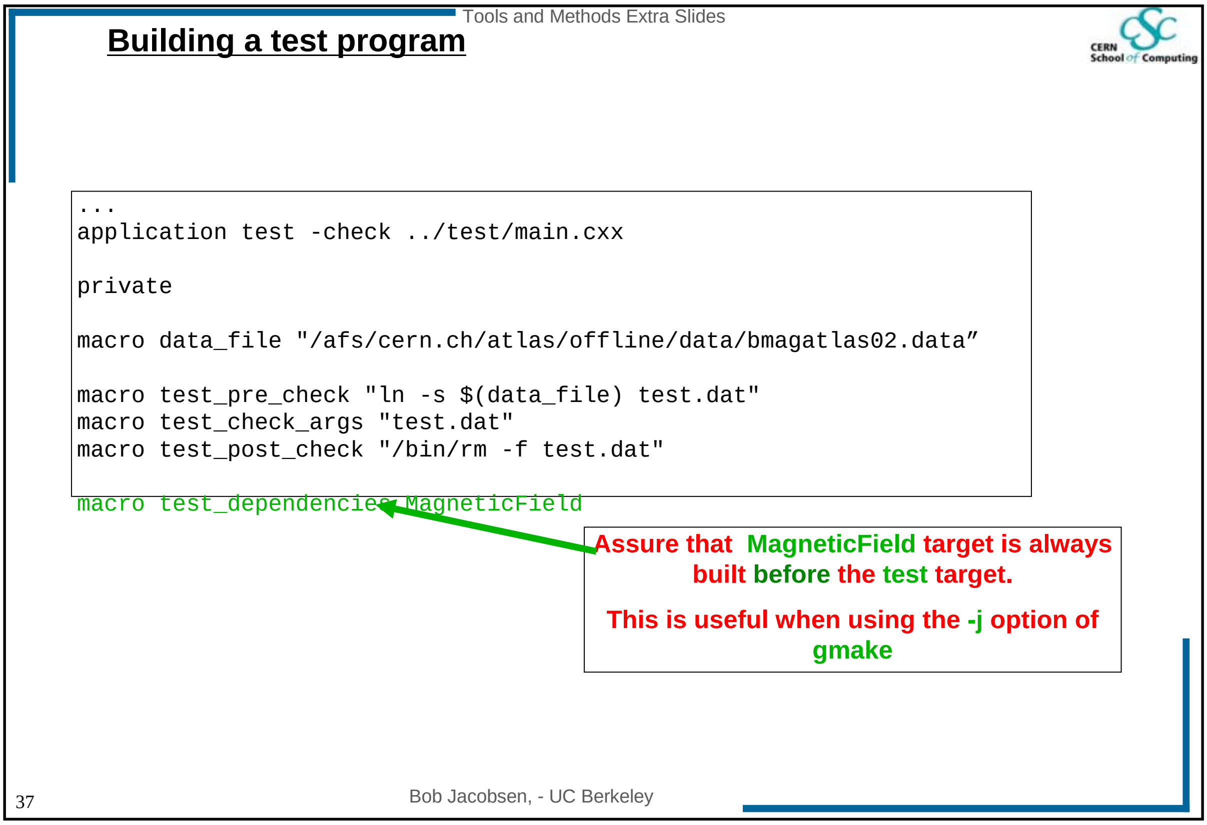
Define data file to be used in the test procedure.



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

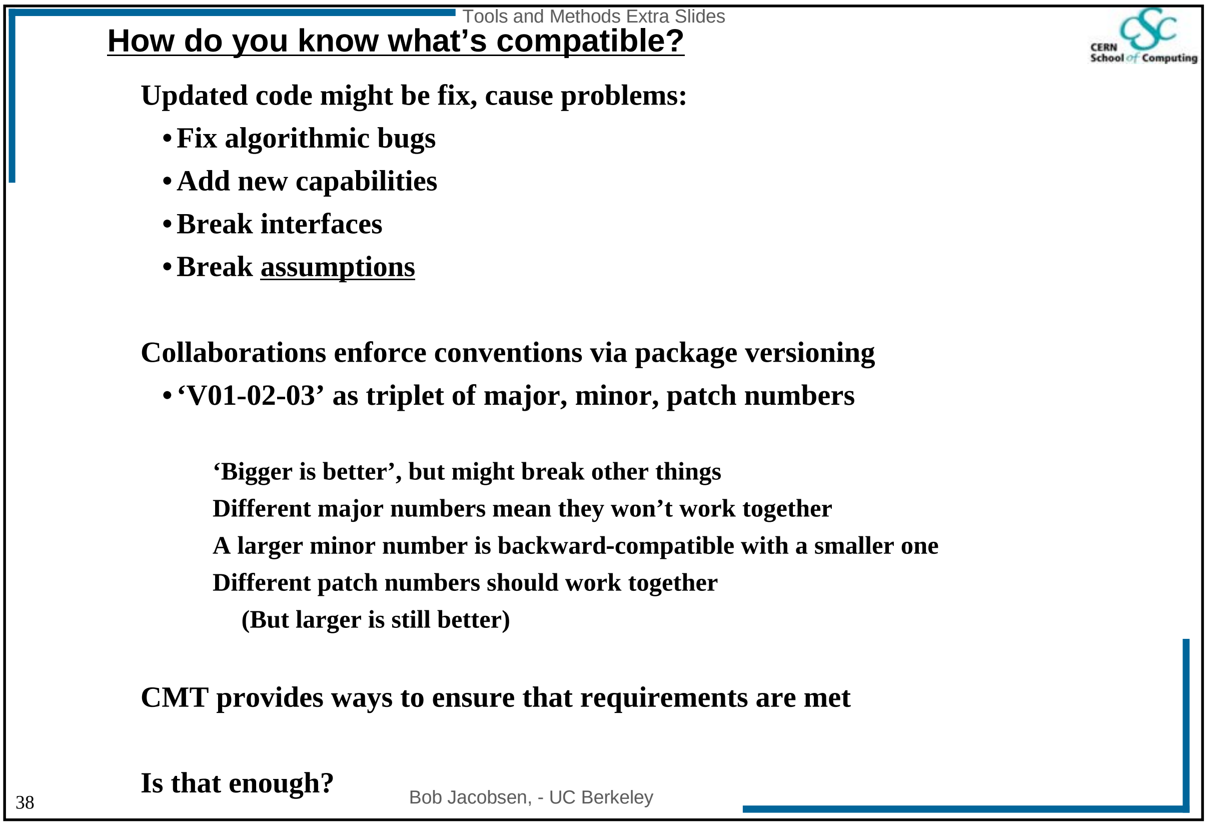


Building a test program

```
...
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?

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?