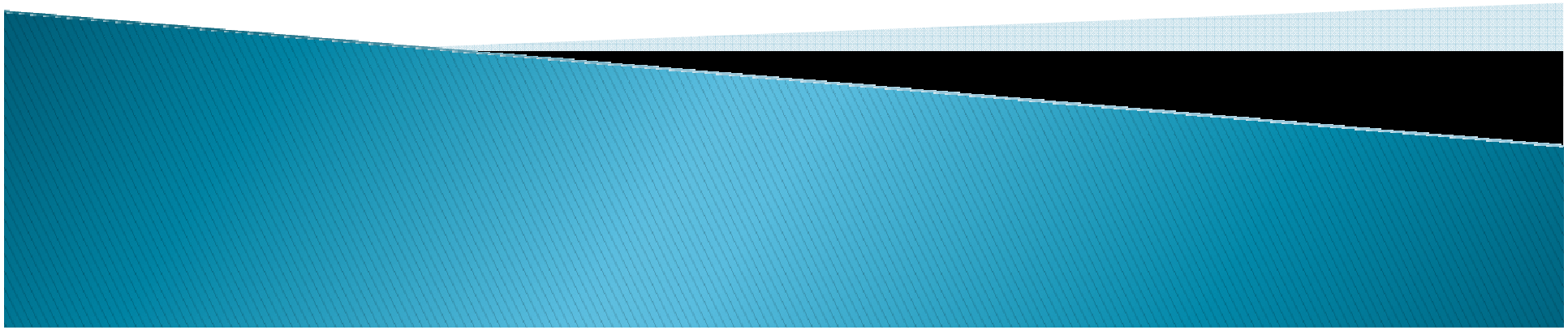
First INFN International School on Architectures, tools and methodologies for developing efficient large scale scientific computing applications

Ce.U.B. – Bertinoro – Italy, 12 – 17 October 2009

Software Physical Design

Pere Mato (CERN)



Outline

- Physical design concepts
- Software development model
- Packaging
- Keeping dependencies under control
- Monitoring and maintaining the software organization

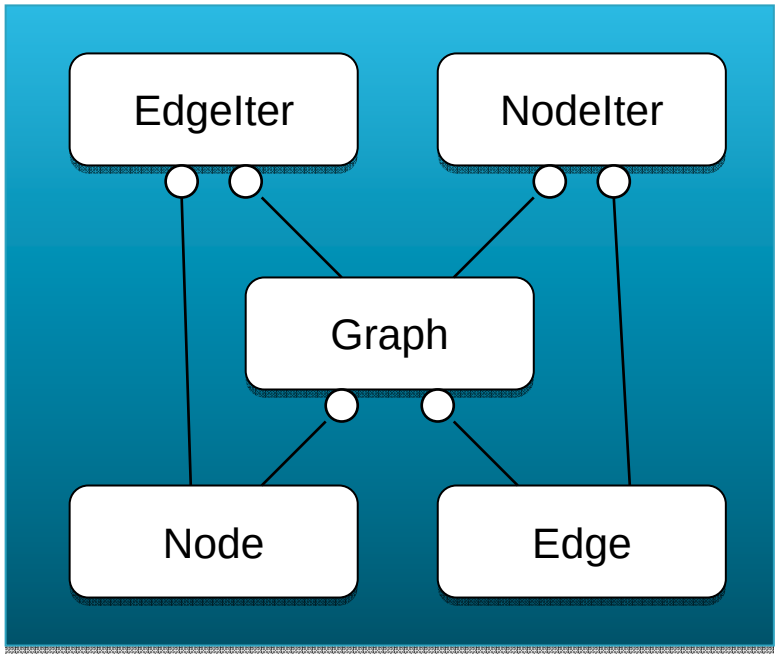
Physical Design Concepts

- Large-scale software development requires more than just logical design issues
  - Distribution of logical entities (classes, functions, etc.) on physical entities (files, directories, etc.)
  - The physical design is the skeleton of the system
- The quality of physical design dictates from the cost of maintenance to run-time performance
  - Additional the potential for re-use
- Component[1] is the fundamental unit of design
- The most important relationship is *DependsOn*
- Logical design addresses architectural issues; physical design addresses organizational issues

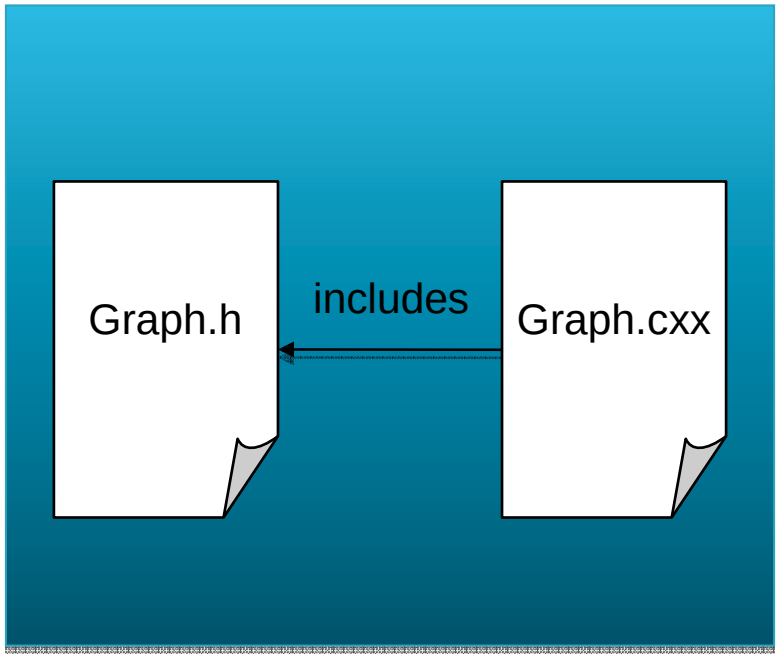
John Lakos, Large-Scale C++ Software design, Addison-Wesley, 1996

Logical vs. Physical View

Logical View



Physical View



Components

- Logical design emphasizes interaction of classes and functions in single seamless space
  - It can be viewed as a 'sea' of classes and functions
  - It does not take into account physical entities such as files and libraries
- A Component would embody a subset of logical design that makes sense to exists as an independent and cohesive unit
- Typically a Component would consists of a single header file (.h) and implementation files (.cxx)

Packages

- Typically in HEP we put each C++ class in a different file (naming convention & convenience)
  - The Lakos's Component concept does not fit completely.
- A Package is a collection of components organized as a physically cohesive unit
- A Package is therefore a collection of Classes and functions that implements some functionality
  - Physically a Package is a collection of header files and implementation files organized in some directory structure
- Package is the basic unit in the HEP software development process

Software Development Process

- How do you create software?
  - Lots of parts: writing, documenting, testing, sharing, fixing,…
  - Usually done by lots of people
- The ‘Process’ is just a big word on how we do this
  - It exists whether you talk about it or not
- Every software production unit (e.g. HEP experiment) follows a process
  - Sometimes undocumented
  - Tools to support the process

Package as Development Unit

- For convenience a Package is developed by one or few developers
  - Concurrent development is essential for large projects
- It is the basic development unit (at least in the HEP communities)
  - It can checked-out and versioned (tagged)
  - It can be tested
  - It can be documented
- Example: ATLAS has ~3000 packages written mainly in C++ and Python (also Fortran, Java, PERL, SQL)

Tools to support the 'Process'

- Code repositories and versioning systems
  - CVS, SubVersion
- Management of versions
  - TagCollectors
- Build and configuration tools
  - Make, CMT, SCRAM
- Nightly build systems
- Test frameworks
  - CppUnit, QMTest
- Documentation
  - Doxygen, lxr, OpenGrog, …
- Distribution
  - Pacman, APT, ..

Physical Elements

- Public Header Files (.h)
- Private Header files (.h)
- Static Libraries (.a)
- Shareable Libraries (.so)
  - Linker Libraries
  - Component Libraries (plug-ins, i.e. no symbols exported)
  - Other modules (e.g. Python extension modules)
- Programs
- Documentation Files (.html, .doc, …)

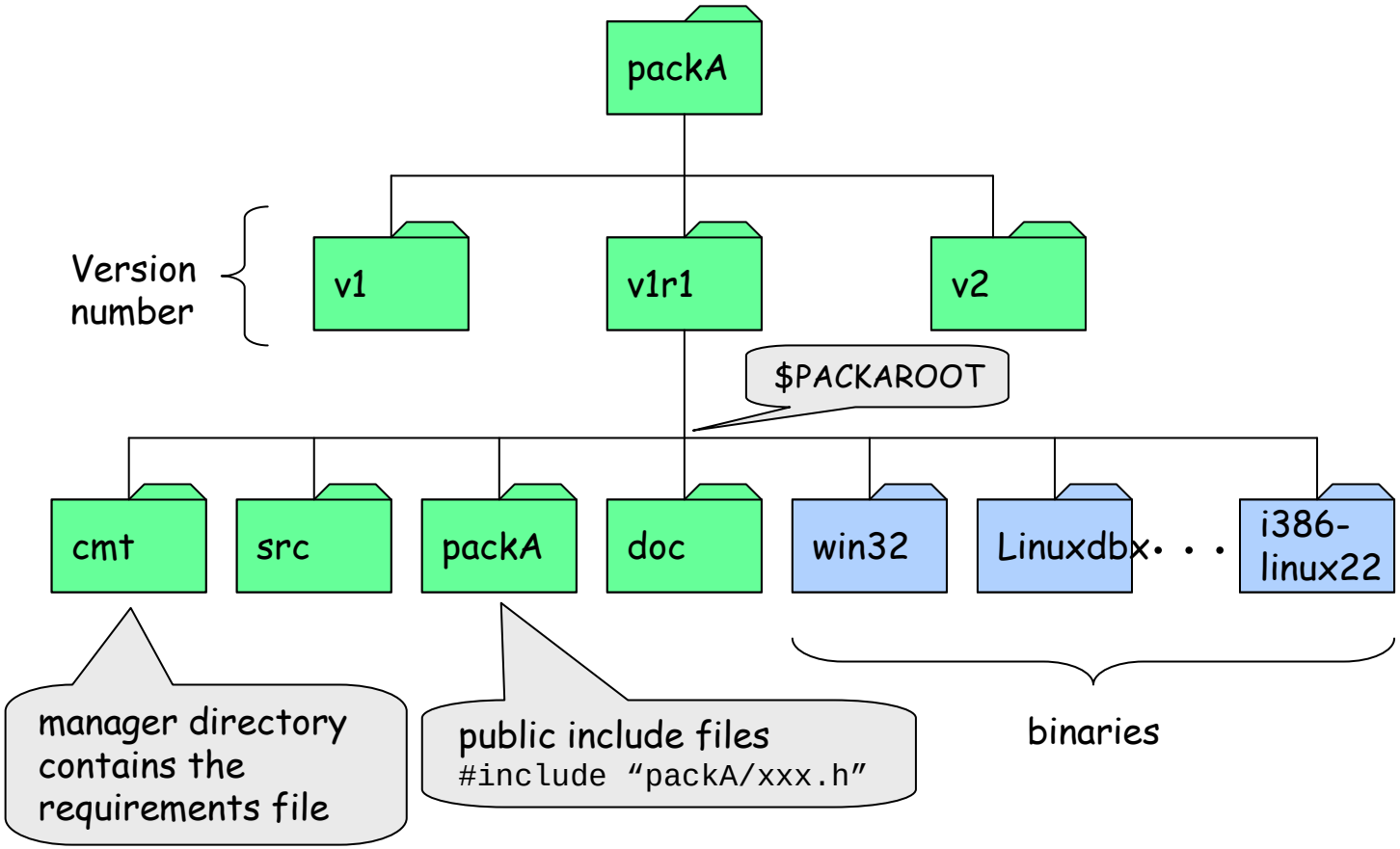
Public Interface of a Package

- Everything declared in its set of public header files
  - Regardless of access privilege (public, protected, private)
  - Any change would cause a re-compilation of clients
- The less information is put on header files the better
  - Favor forward declarations of types used as references and pointers

Major Design Rules

- Avoid definitions with external linkage in .cxx files that are not declared explicitly in the corresponding .h file
  - Define exclusively what is declared (no backdoors)
- Avoid accessing a definition with external linkage in another package via local declaration
  - Include the .h file for that package

Typical Package Structure

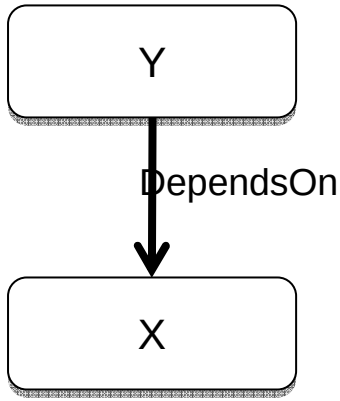


Package Products

- Linker Libraries
  - Are traditional libraries. They export a number of symbols
- Component or plug-in libraries
  - These libraries are loaded at run-time on demand by the application (framework)
  - Typically they do not export any symbol. In some cases a single global one
- Programs, Tests
  - Either direct executables or plug-ins
- Documentation
- Additional framework files
  - Configuration files, plugin databases, etc.

Package Dependencies

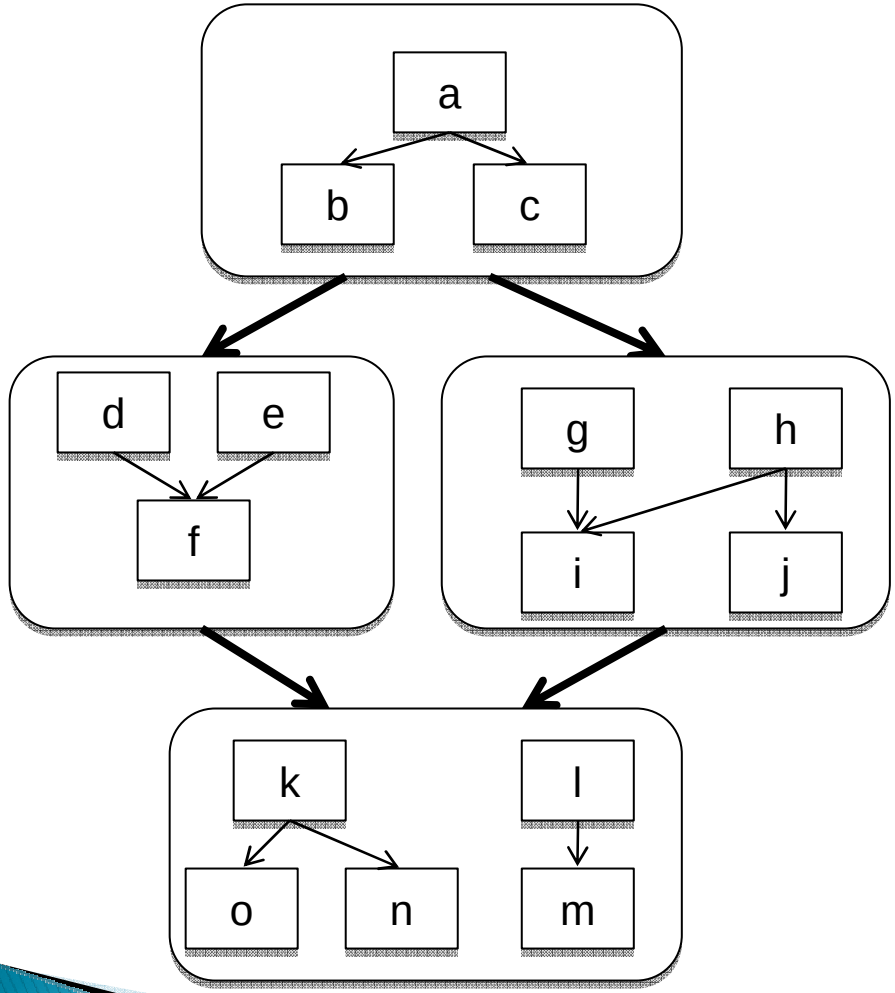
- A package Y *DependsOn* a package X if X is needed in order to compile or link Y
  - Compile-time dependency if one or more .h files in X are needed for compilation
  - Link-time dependency if one or more libraries in X are needed for linking
  - Run-time dependency if a program/library in package Y requires X for running
- In general compile-time dependency implies link-time dependency and this implies run-time dependency
  - Templates defeats this general rule!
- The DependsOn relation is transitive

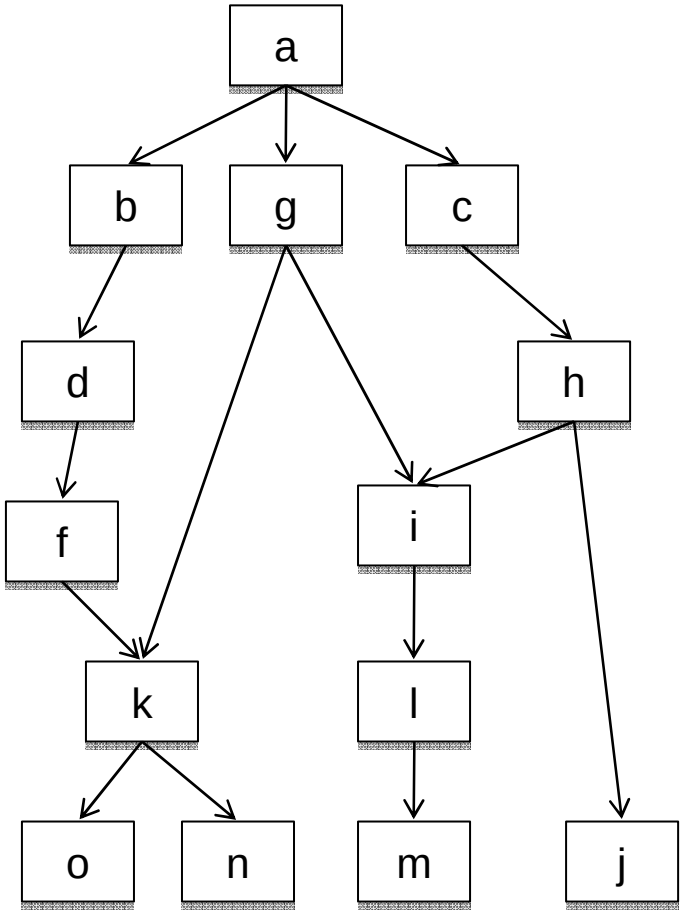


Package Dependencies (2)

- A package defining a function will have a physical dependency to any other package defining a type used in the function
- The logical relationship HasA and IsA translates into a physical dependency
- Dependencies limit
  - flexibility
  - ease of maintenance
  - reuse of components or parts
- Dependency management tries to control dependencies

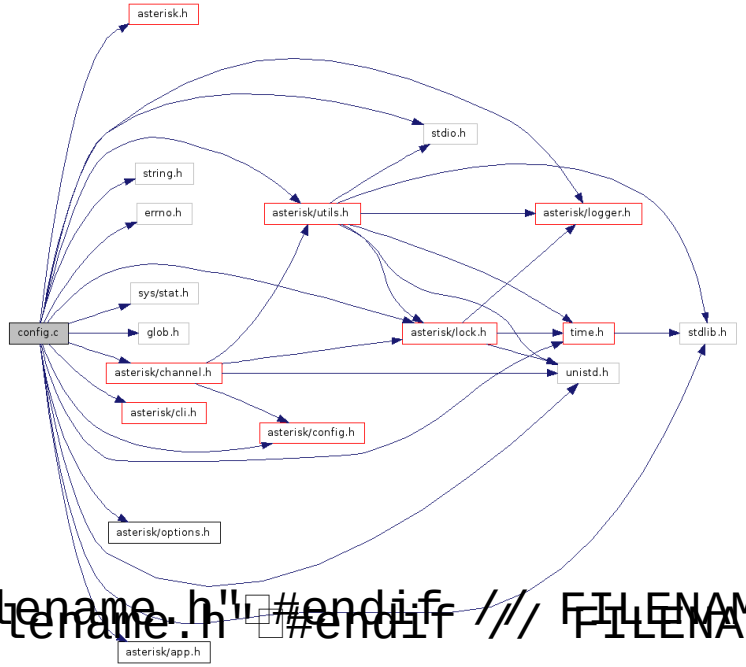
Packages Dependencies (3)





Compile-time dependencies

- Cyclic dependencies would prevent building the package. End of story.
- Tools such as Doxygen allows to monitor dependencies
- Thinning header files will speedup building process
- External include guards, or redundant include guards, were suggested by John Lakos




```
#ifndef FILENAME_H_□#include "Filename.h"□#endif /// FILENAME_H_
```

Link(Load)-time dependencies

- The use of dynamic libraries converts link-time dependencies to load-time ones
  - Static libraries are not in fashion nowadays
- Tools such ldd (depends.exe on Windows) allows to monitor link dependencies
- Performance is strongly affected by the number and the size of dependent libraries
  - Interest to keep the them under control
- Reduce the number of needed libraries
  - re-packaging, re-engineering
- Remove unnecessary libraries
  - Control package dependencies; use --as-needed flag

Compile and Link Times

- Compile and link times are unproductive
- In a project with N modules compile and link time can grow like N^2 (assuming every package is tested) when dependencies are not controlled
- Loss of productivity
- Long turnaround times → slow development
- Dependency management essential in large projects

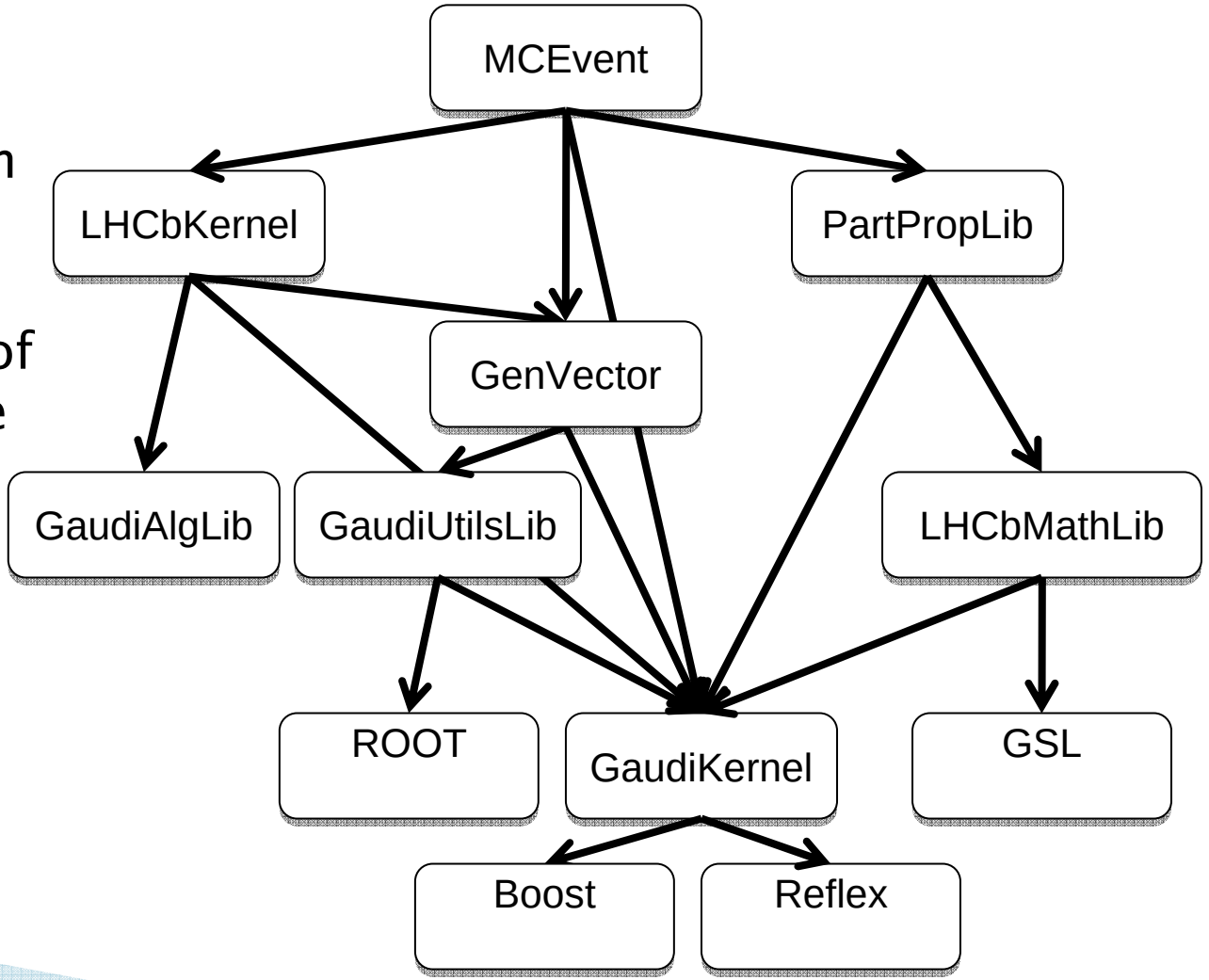
Realistic Example

```
>>> ldd libMCEvent.so
        libdl.so.2 =>
        libLHCbKernel.so =>
        libPartPropLib.so =>
        libGaudiKernel.so =>
        libpthread.so.0 =>
        libGenVector.so =>
        libstdc++.so.6 =>
        libgcc_s.so.1 =>
        libc.so.6 =>
        libGaudiAlgLib.so =>
        libm.so.6 =>
        libLHCbMathLib.so =>
        libReflex.so =>
        libboost_thread-gcc34-mt-1_39.so.1.39.0 =>
        libboost_system-gcc34-mt-1_39.so.1.39.0 =>
        libboost_filesystem-gcc34-mt-1_39.so.1.39.0
=>
        libCore.so =>
        libCint.so =>
        libGaudiUtilsLib.so =>
        libboost_regex-gcc34-mt-1_39.so.1.39.0 =>
        libgsl.so.0 =>
        libgslcblas.so.0 =>
        librt.so.1 =>
        libcrypt.so.1=>
        libList.so => l
        libMatrix.so
        libMathCore.so =>
```

- libMCEvent.so is a library for MC event classes
  - A priory it should not depend on Boost, ROOT, Math, GSL, etc.
- The problem is that it depends on GaudiKernel and others that these depend on Boost, ROOT, etc.

Understanding Dependencies

- The 22 classes in the MCEvent package DependOn only 4 packages
- Only few classes/functions of these packages are really needed
- These initial dependencies brings the rest
- Is this a real problem?

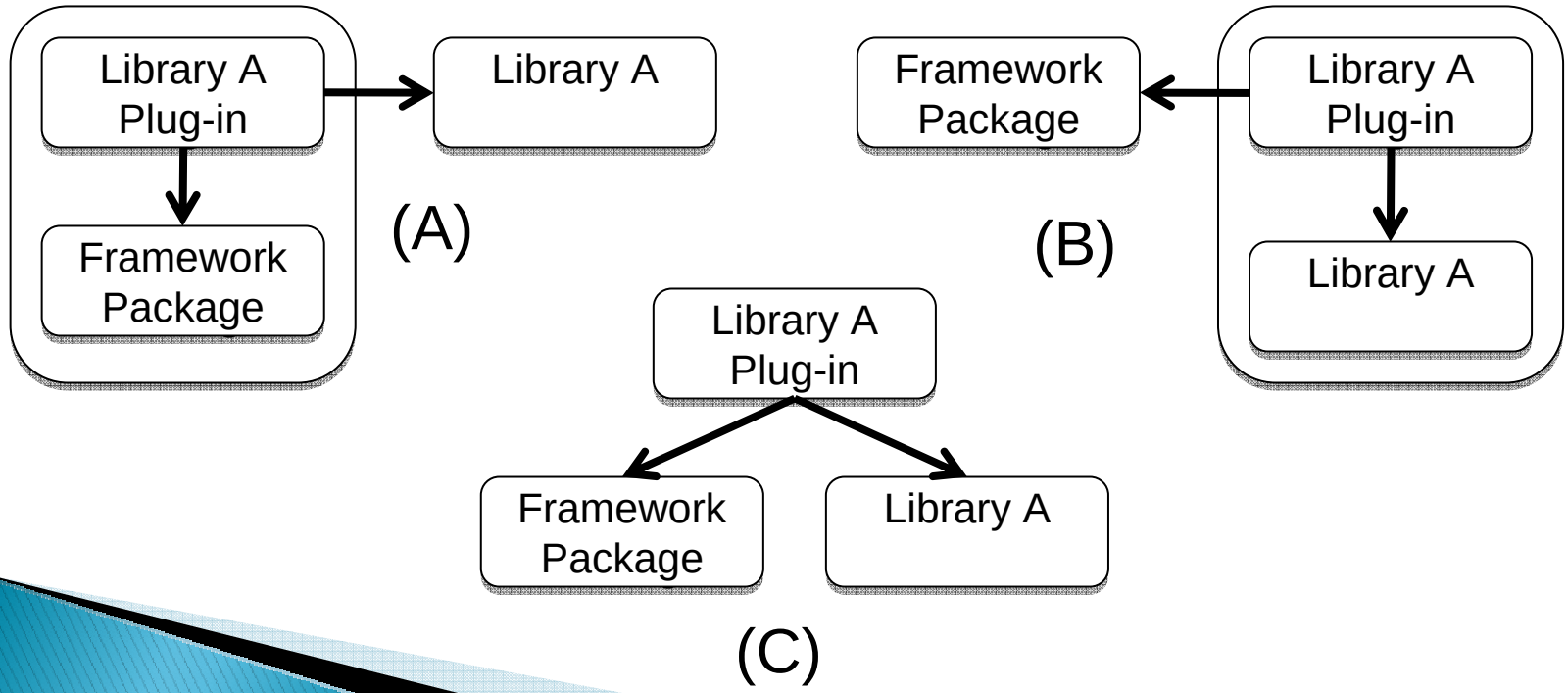


Run-time dependencies

- These dependencies are due typically to the plug-in mechanism, [Reflex] dictionary loading, Python extension modules, etc.
  - Frameworks make extensive use of run-time dependencies
- Moving compile and link time dependencies to run-time dependencies is not a bad move
  - Only needed functionality will be loaded
- Packaging and installation of 'plug-ins' is non-trivial

Plugins

- At least three possibilities for packaging plug-ins
- (C) is the one that creates less coupling
- (A) and (B) forces a dependency between the library and the framework



Configuration Tools

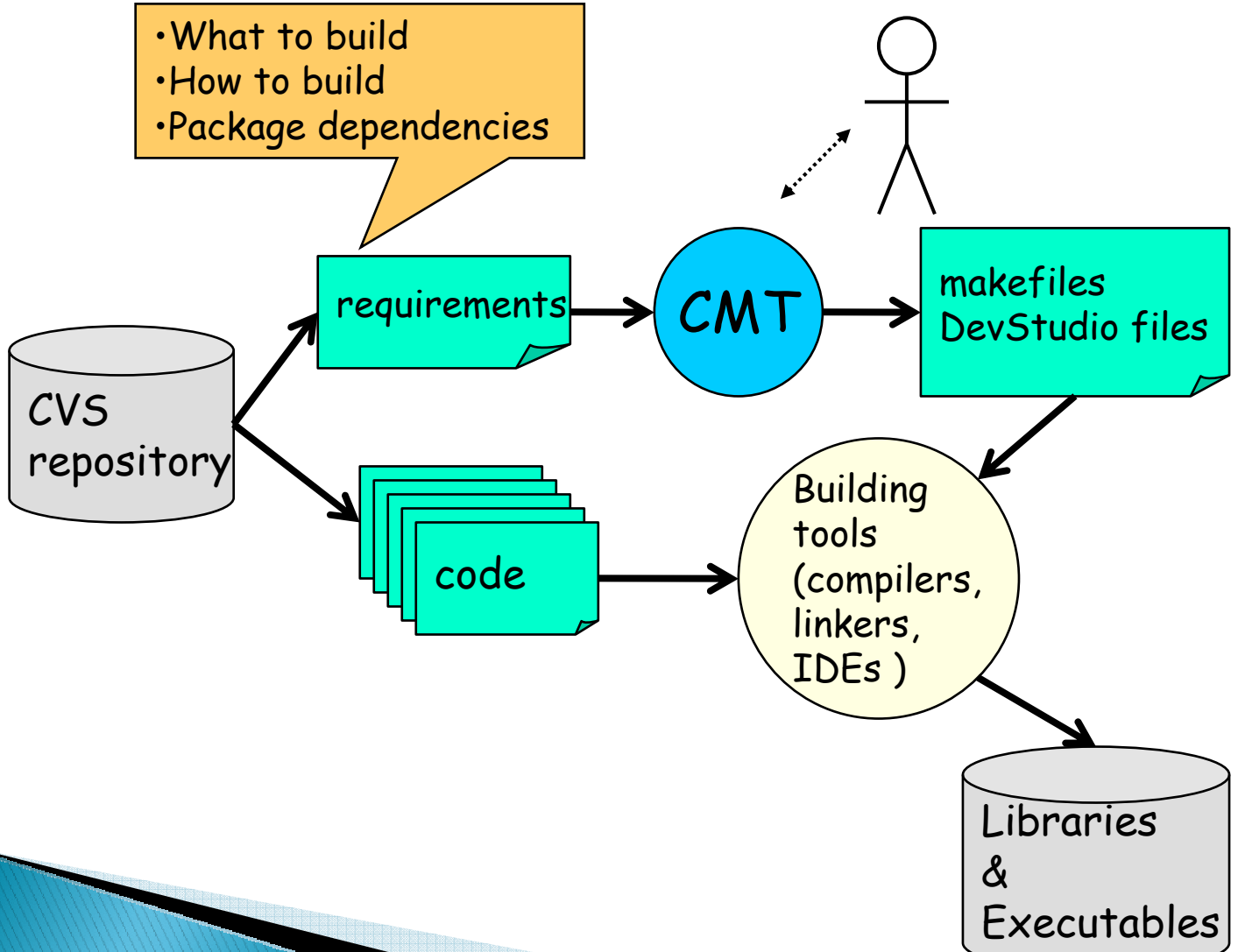
- To manage the the dependencies between packages and to facilitate the building of packages, experiments are using configuration and build tools
  - Remember ATLAS has ~3000 packages :-0
  - CMT (ATLAS, LHCb), SCRAM (CMS), SRT (BaBar)
- These tools can typically
  - Find inconsistencies
  - Create include and library options
  - Connect build constituents

  ➔ Automation of the process

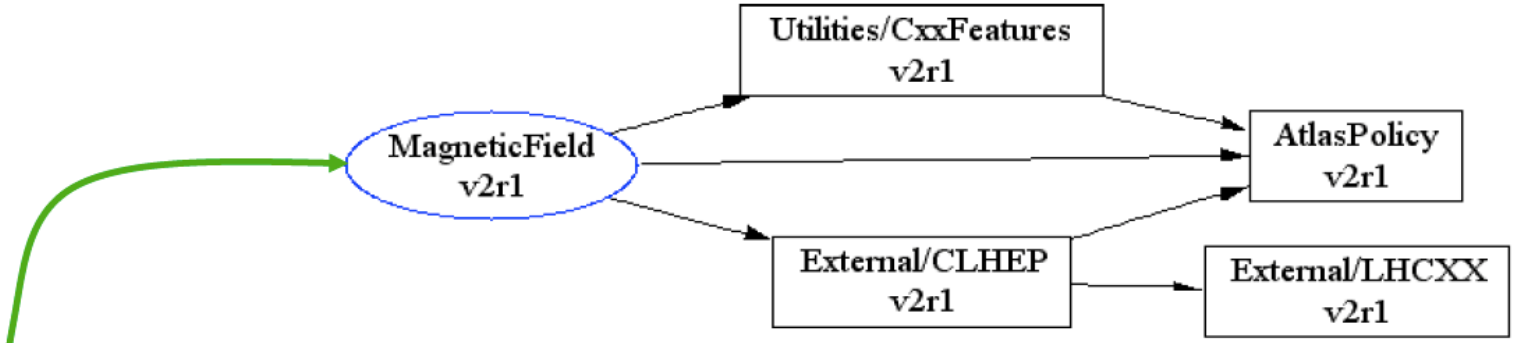
CMT

- Configuration Management Tool written by C. Arnault (LAL, Orsay)
  - It is based around the notion of *Package*
  - Provides a set of *tools for automation* the configuration and building packages
    - A variety of products: libraries (linker, plugins), executables, documentation, etc.
  - It has been adopted by LHCb, ATLAS (other experiments are also using it)

Using CMT


•What to build
•How to build
•Package dependencies
requirements
CMT
makefiles
DevStudio files
CVS
repository
code
Building
tools
(compilers,
linkers,
IDEs)
Libraries
&
Executables

CMT requirements file

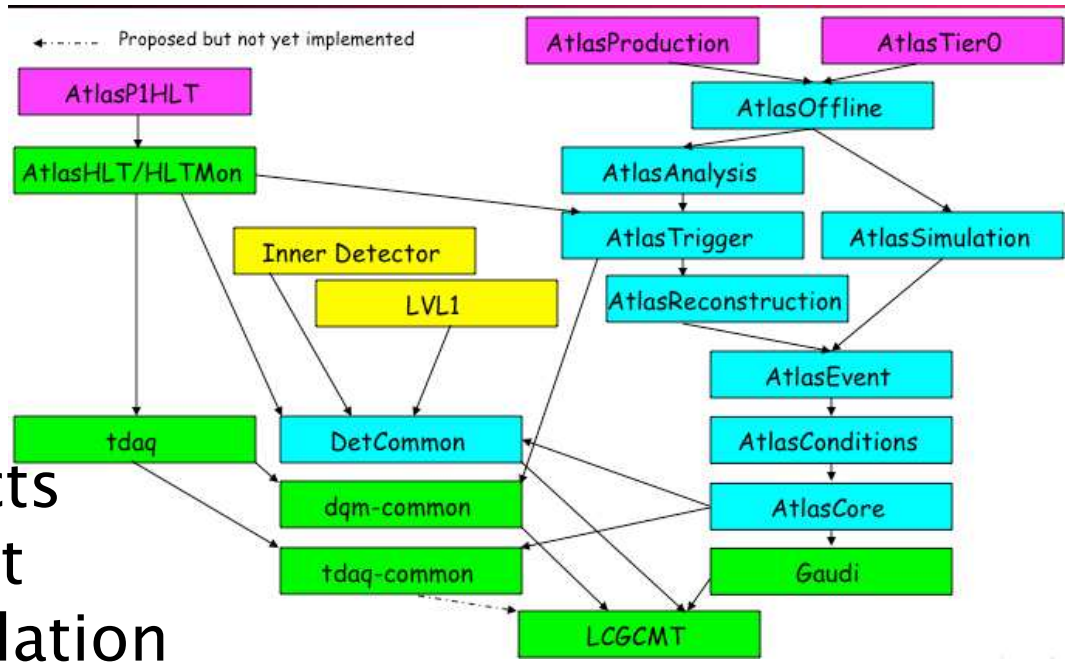



```
package MagneticField

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)

Grouping Packages

- Managing and releasing many packages is complex
  - Integrating and validating releases may take very long
- Some experiments are grouping 'packages' into 'projects'
- A 'project' is basic unit of release
- The same way packages facilitated concurrent development; projects facilitates concurrent integration and validation

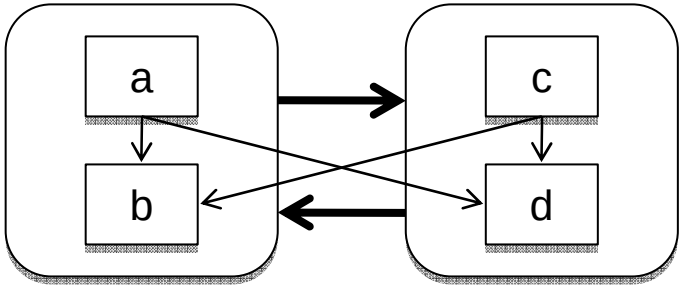


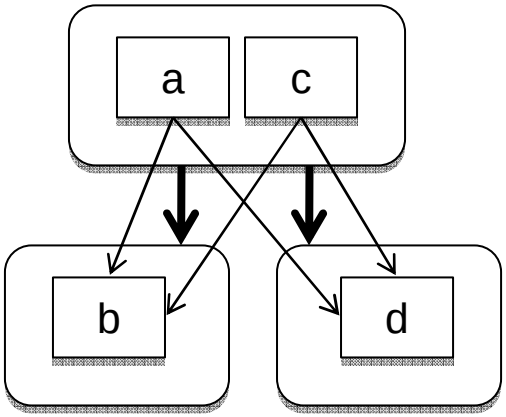
ATLAS Project Structure

Software Release

- Experiments do not release individual packages
  - Each individual package is ‘tagged’ by developer
- Experiments release complete ‘projects’
  - The release candidate of a project is made of a collection of ‘tags’ for each package that constitutes the project
  - Tag collector tools helping here
- Obviously the release order is opposite to the DependsOn relationship
  - The version of the top-level project fixes the version of each dependent project and each package within the project
- Not all the software system needs to be released at once

Package Levelization Techniques

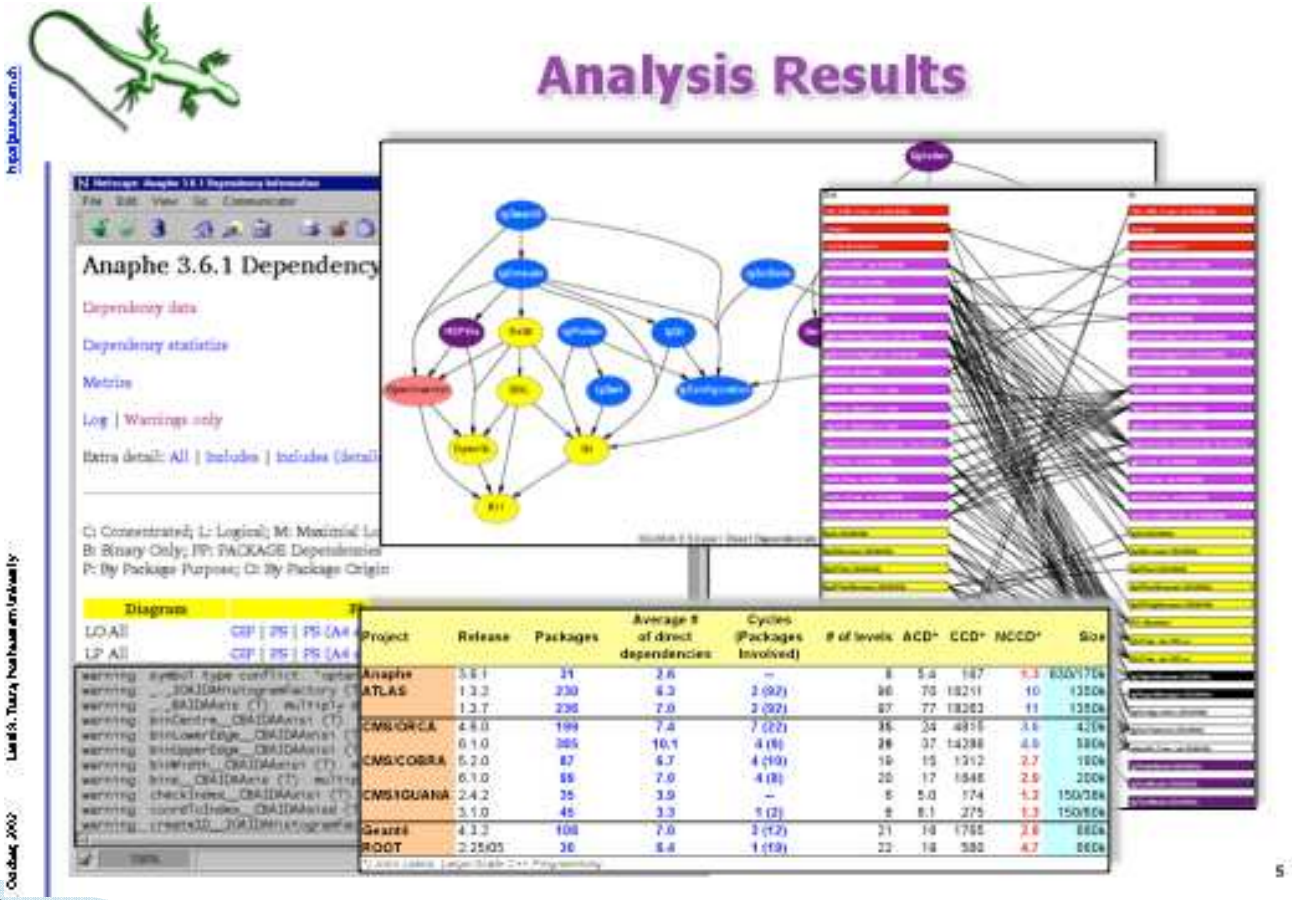




- Example: avoid cyclic dependencies among packages

Tools to Check Dependencies

- Ignominy (L. Tuura)



Ignominy

Dependency Analysis

- Ignominy scans...
  - Make dependency data produced by the compilers (*.d files)
  - Source code for #includes (resolved against the ones actually seen)
  - Shared library dependencies ("ldd" output)
  - Defined and required symbols ("nm" output)
- And maps...
  - Source code and binaries into packages
  - #include dependencies into package dependencies
  - Unresolved/defined symbols into package dependencies
- And warns... about problems and ambiguities (e.g. multiply defined symbols or dependent shared libraries not found)
- Produces a simple text file database for the dependency data

More Principles

- When adding a new component to a package, both the logical and physical characteristics of the component should be considered
- Minimizing the number and size of exported header files enhances usability
- A [binary compatible] patch must not affect the internal layout of any existing object (i.e. no change in any header file)

  ➔ Physical design is

Summary

- A large software system is organized as Classes ➔Components ➔ Packages ➔ Projects
  - Keep complexity under control
  - Facilitating concurrent development
- Physical dependencies limits strongly the performance and overall quality of the system
  - Flexibility, ease of maintenance, reuse of components or parts
- Tools to monitor and optimize dependencies are essential
- The software architect is also responsible for the physical design

References

- John Lakos, Large-Scale C++ Softare Design, Addison-Wesley, 1996