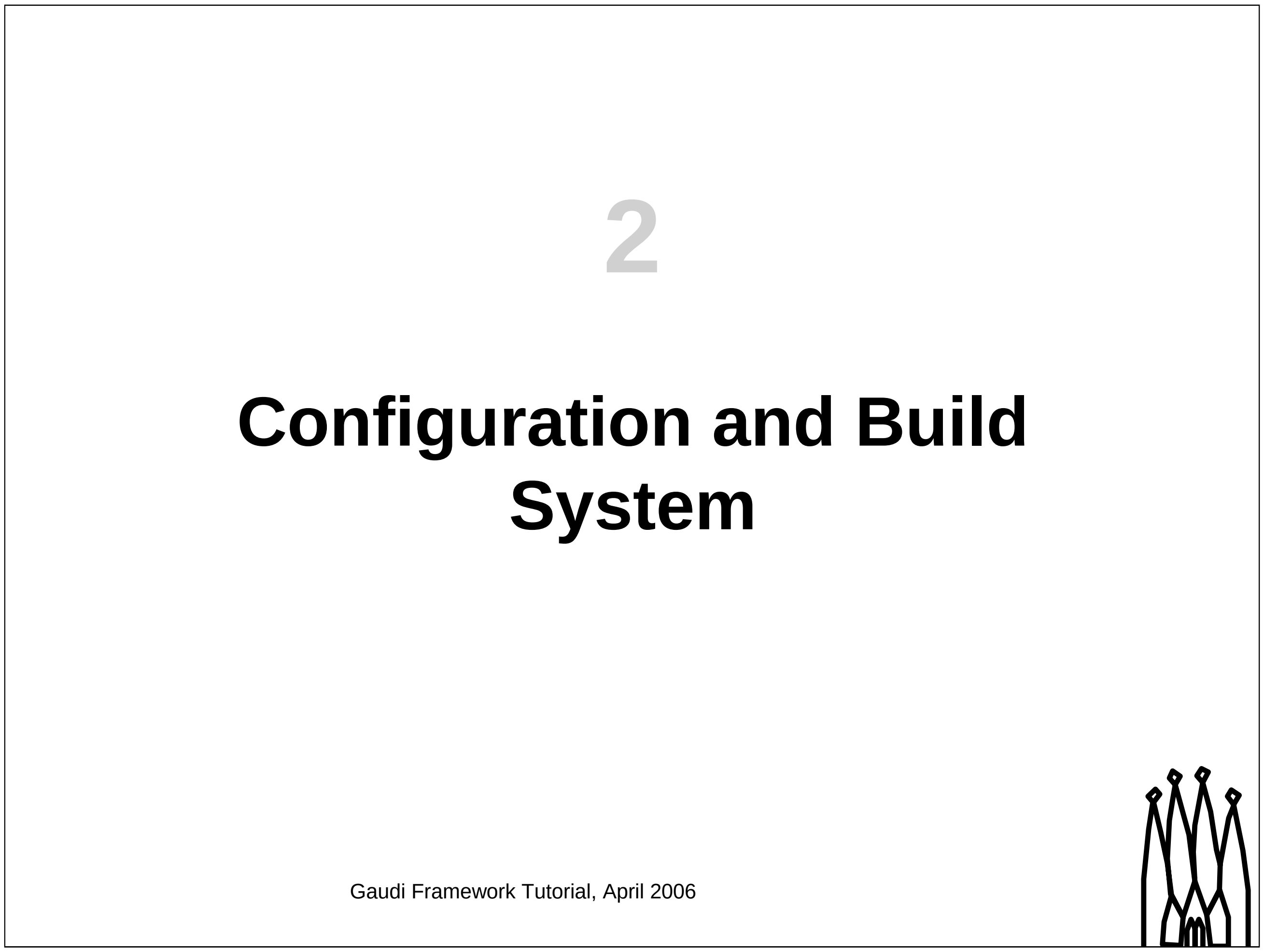


2

# Configuration and Build System

Gaudi Framework Tutorial, April 2006

**Schedule:**

| Timing | Topic |
|---|---|
| 20 minutes | Lecture |
| 10 minutes | Practice |
| 30 minutes | Total |

## Objectives

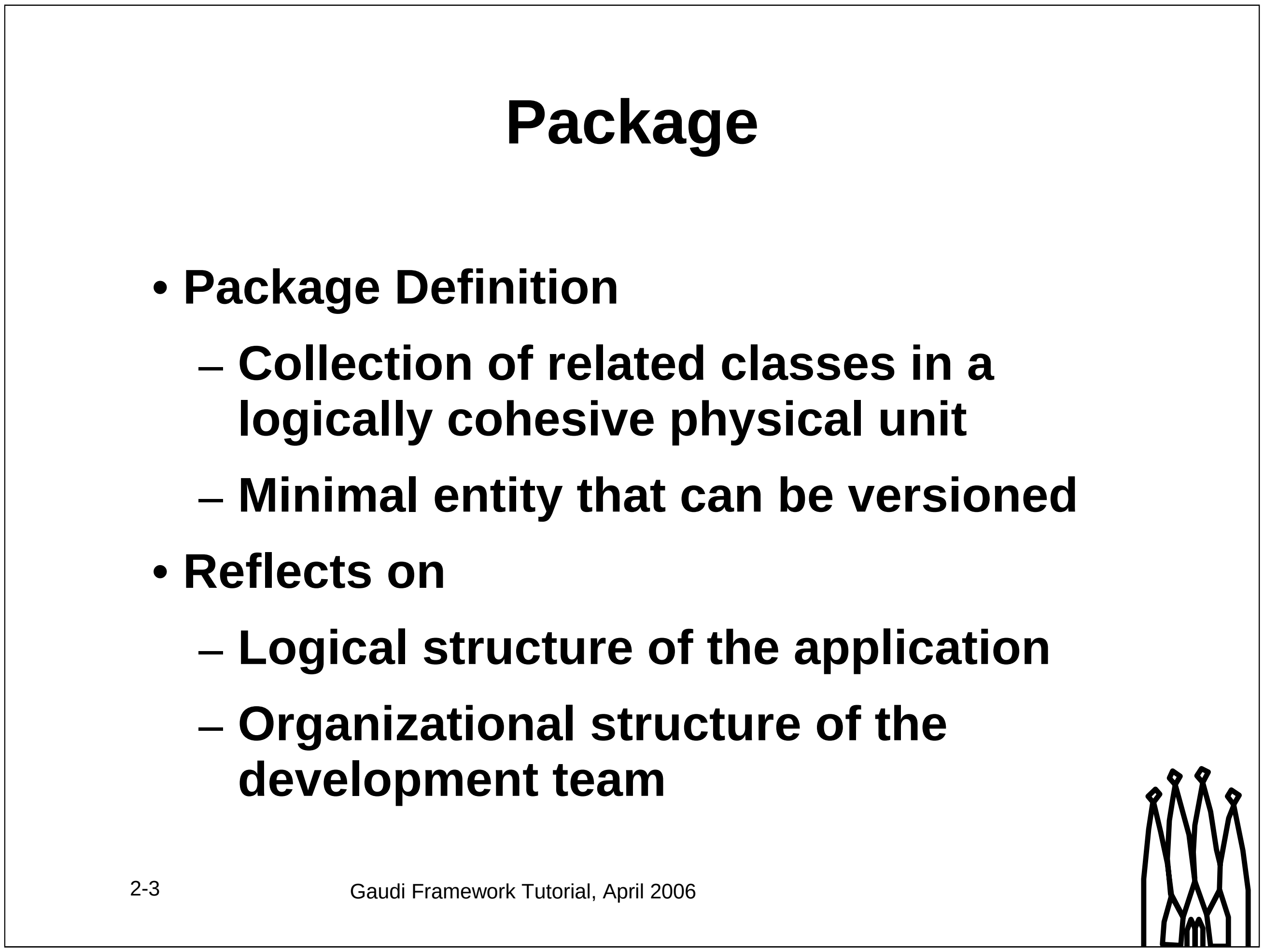
**After completing this lesson, you should be able to:**

- **Understand the LHCb Configuration Management**
- **Get a copy of a package from the LHCb code repository**
- **Know how to re-build libraries and programs**

### Lesson Aim

Understanding and being able of using the basic commands of the configuration and build system is a pre-requisite for the rest of the Tutorial.

The commands introduced in this lesson would allow us to get a copy of the packages that we are going to use for the rest of the Tutorial.



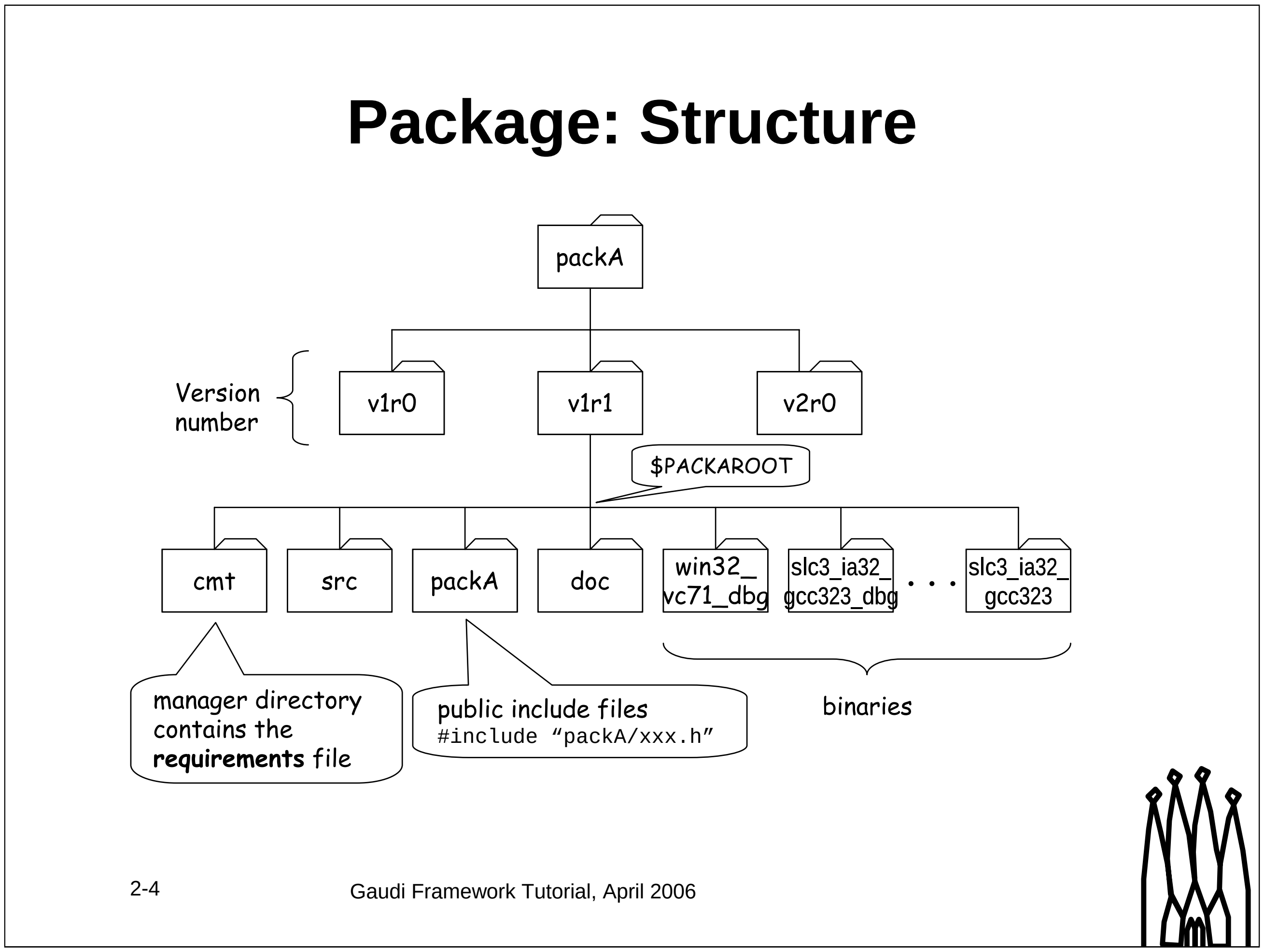
## Package

- Package Definition
  - Collection of related classes in a logically cohesive physical unit
  - Minimal entity that can be versioned
- Reflects on
  - Logical structure of the application
  - Organizational structure of the development team

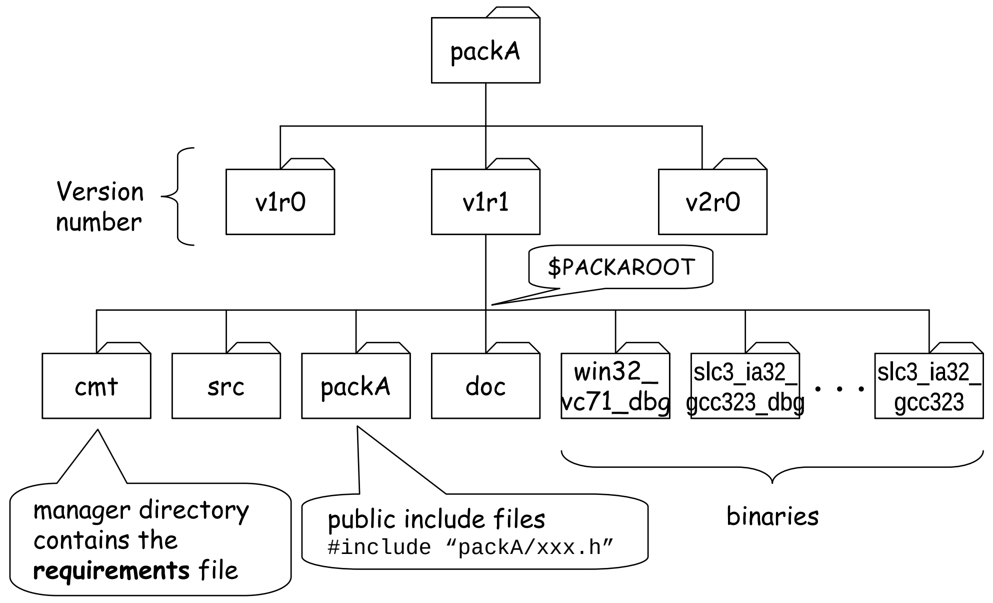
2-3 Gaudi Framework Tutorial, April 2006

**Package Definition**

We define a “package” as a collection of related classes and as being the minimal entity that can be versioned and built. From this definition is clear that the configuration and build system we are using only deals with “packages” and not with individual classes or subroutines.



## Package: Structure



2-4 Gaudi Framework Tutorial, April 2006

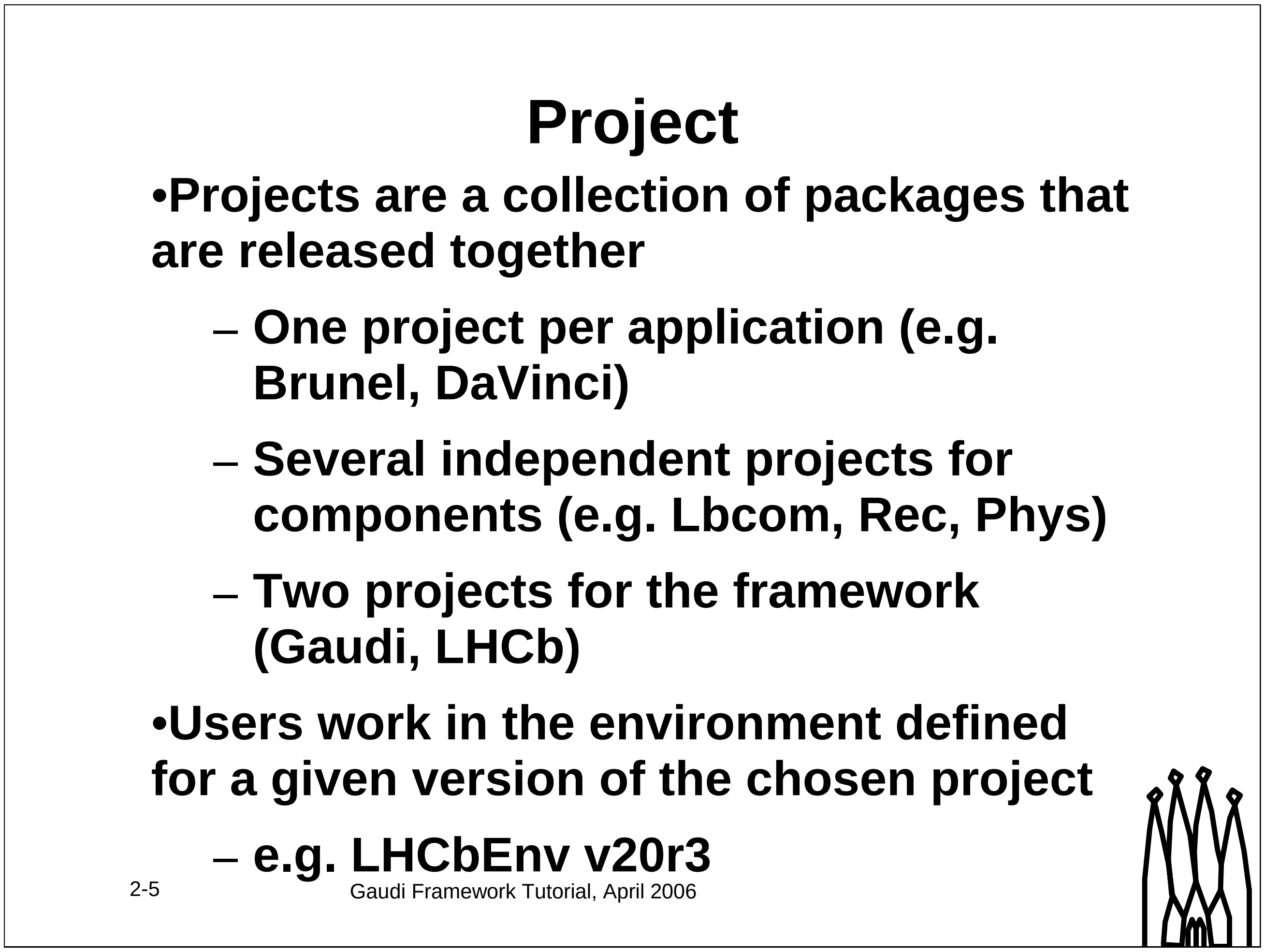
### Package Structure

All packages in LHCb follow the structure shown in the viewgraph.

- Package version name follow the convention “v<version number>r<release number>p<patch number>” or “v<version number>r<release number>d<date>”. Versions with the same v number should be considered compatible.
- The /cmt directory is mandatory (see later with CMT)
- The source files are located in /src and the exported header files are in /packA. In that way it is visible from which package a header is included.
- The /doc directory contains the documentation for the package. It is mandatory to have a file called “release.notes” where we keep the changes on the package up to date.
- A number of binary (platform & configuration dependent) directories will exists in each package.

### Package Groups (Hats)

We have organized the packages in LHCb by putting together all related packages into a package group. Examples are: Event, Det, Ex, etc.



## Project

- Projects are a collection of packages that are released together
  - One project per application (e.g. Brunel, DaVinci)
  - Several independent projects for components (e.g. Lbcom, Rec, Phys)
  - Two projects for the framework (Gaudi, LHCb)
- Users work in the environment defined for a given version of the chosen project
  - e.g. LHCbEnv v20r3

2-5 Gaudi Framework Tutorial, April 2006

**LHCb Software Project:**

Projects are the collection of packages that are released together. A given version of a project contains a single version of all its constituent packages. This includes the sources and the binaries for all the supported platforms. Each new release of a project implies a complete rebuild from sources of the constituent packages.

We have defined two projects for the framework:

- Gaudi contains all the packages of the (experiment neutral) Gaudi distribution
- LHCb contains all the LHCb specific common software (e.g. event model, detector description etc.)

Component packages (i.e. packages that contain algorithms and tools built on the framework), reside in one of several independent projects broadly classified by functionality. We currently have the following component projects, which are all built on top of the framework projects:

- Lbcom contains packages shared by all applications (e.g. general purpose Rich tools)
- Rec contains reconstruction packages (e.g. tracking pattern recognition)
- Phys contains physics analysis packages (e.g. physics selections)
- Online contains packages needed for running in the online farm (e.g. Gaucho)

There is then one project per application (e.g. Brunel, DaVinci) which contains the application itself and possibly other packages very specific to the application, and uses one or more of the component packages.

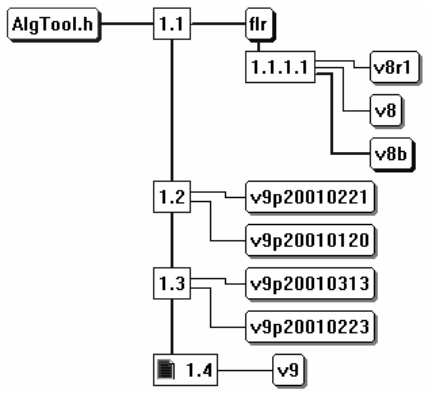
Users must work inside a given project environment. They must choose the project and the version. The ProjectEnv script sets up all the environment variables required to work in the project. It is invoked as <project>Env <version>, e.g. LHCbEnv v20r3

# CVS

**Version Control System**

- **Record the history of your source files**
- **Helps you if you are part of a group of people working on the same project.**

**(Repository, Module, File, Version, Tag)**



2-6 Gaudi Framework Tutorial, April 2006

**Basic Description of CVS**

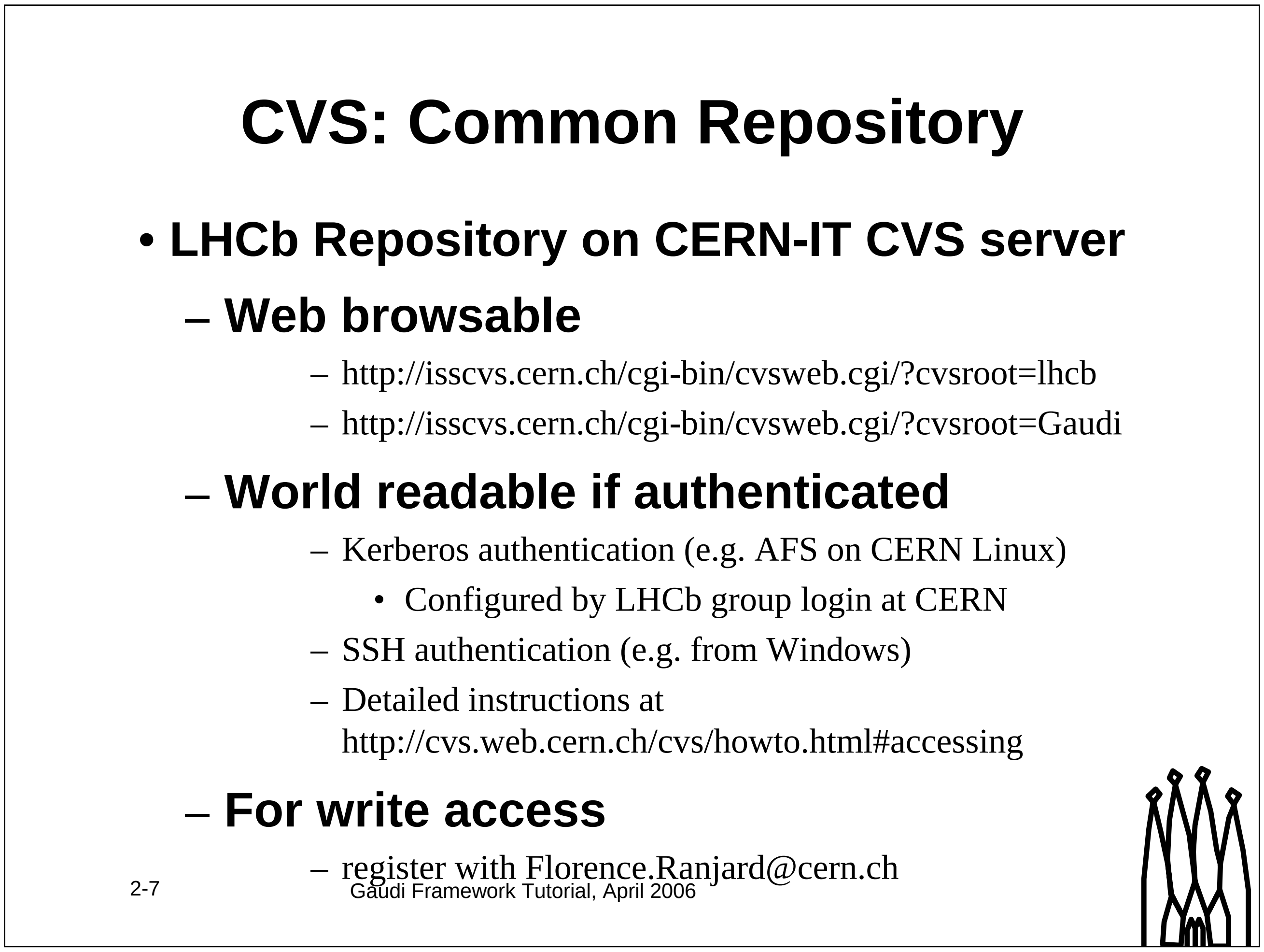
CVS is a system that lets groups of people work simultaneously on groups of files (for instance packages).

It works by holding a central `repository' of the most recent version of the files. You may at any time create a personal copy of these files by `checking out' the files from the repository into one of your directories. If at a later date newer versions of the files are put in the repository, you can `update' your copy.

You may edit your copy of the files freely. If new versions of the files have been put in the repository in the meantime, doing an update merges the changes in the central copy into your copy.

When you are satisfied with the changes you have made in your copy of the files, you can `commit' them into the central repository.

When you are finally done with your personal copy of the files, you can `release' them and then remove them



## CVS: Common Repository

- LHCb Repository on CERN-IT CVS server
  - Web browsable
    - http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/?cvsroot=lhcb
    - http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/?cvsroot=Gaudi
  - World readable if authenticated
    - Kerberos authentication (e.g. AFS on CERN Linux)
      - Configured by LHCb group login at CERN
    - SSH authentication (e.g. from Windows)
    - Detailed instructions at http://cvs.web.cern.ch/cvs/howto.html#accessing
  - For write access
    - register with Florence.Ranjard@cern.ch

### CVS Repository

**Code Repository Location**

The LHCb code repository is located in AFS and accessed via the CERN-IT CVS server.

The repository is web browsable at http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/?cvsroot=lhcb

**Using the CVS server**

Read access to the browser is available to any authenticated client, either via Kerberos 4 or SSH.
Write access can be obtained by registering with Florence.Ranjard@cern.ch

If Kerberos 4 is available in your platform (e.g. AFS in CERN RedHat Linux distribution), set CVSROOT = :kserver:isscvs.cern.ch:/local/reps/lhcb
This is done automatically in the LHCb group login at CERN

If not, you must use SSH.

Detailed instruction for configuring Kerberos access from outside CERN, and SSH access from both Linux and Windows, are available at:
http://cvs.web.cern.ch/cvs/howto.html#accessing

**CVS packages**

From time to time you may need to create a new package in CVS. Before doing so you should discuss your proposed packaging with one of the librarians or program maintainers. You can then follow the detailed instructions at http://cern.ch/lhcb-comp/Support/CMT/CMTnewpackage.htm

## CMT

**Configuration Management Tool written by C. Arnault (LAL, Orsay)**

- **It is based around the notion of *Package***
- **Provides a set of *tools for automating* the configuration and building packages**
- **It has been adopted by LHCb (other experiments are also using it)**

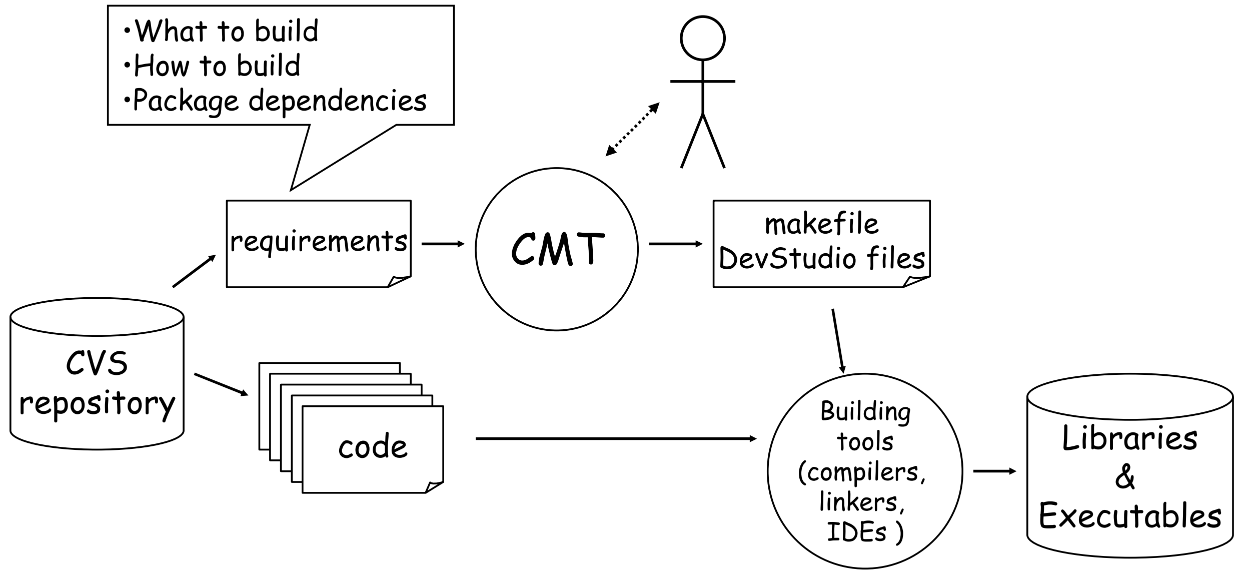
2-8 Gaudi Framework Tutorial, April 2006

### CMT introduction

CMT is an attempt to formalize software production and especially configuration management around a *package*-oriented principle. The notion of *packages* represents hereafter a set of software components (that may be applications, libraries, documents, tools etc...) that are to be used for producing a *system* or a *framework*. In such an environment, several persons are assumed to participate in the development and the components themselves are either independent or related to each other.

The environment provides conventions (for *naming* packages, files, directories and for *addressing* them) and tools for *automating* as much as possible the implementation of these conventions. It permits to *describe* the configuration requirements and automatically deduce from the description the effective set of configuration parameters needed to operate the packages (typically for *building* them or *using* them).

## How we use CMT



2-9 Gaudi Framework Tutorial, April 2006

### How we use CMT

The user interacts mainly with the CMT tools to configure and build the packages. The instructions on what to build, how to build and dependencies are located in a single text file called *requirements*. Very often the user needs to edit this file.

From the *requirements*, CMT is able to automate the creation of the makefiles (or Visual Studio projects) required for building the different package constituents (libraries, programs, documentation, etc).

Note that CMT takes care of generating all files needed to make a package (e.g. dependencies makefiles). When you type “make”, what actually happens is that make calls CMT, which regenerates all the necessary makefiles, before calling the real make.

## CMT: Requirements file

```
package Main
version v5r3
branches doc job options cmt

use Components    v7r* Tutorial
use GaudiSvc      v*               -no_auto_imports
use GaudiPoolDb   v*               -no_auto_imports
use PackedEvent   v*   Event       -no_auto_imports
use GenEvent      v*   Event       -no_auto_imports
use RootHistCnv   v*               -no_auto_imports
use ParamFiles    v*               -no_auto_imports
use GaudiConf     v*               -no_auto_imports

#==> Build the main program
application Main "$(GAUDICONFROOT)/src/GaudiMain.cpp"
apply_pattern application_path
```

**Some basic keywords**

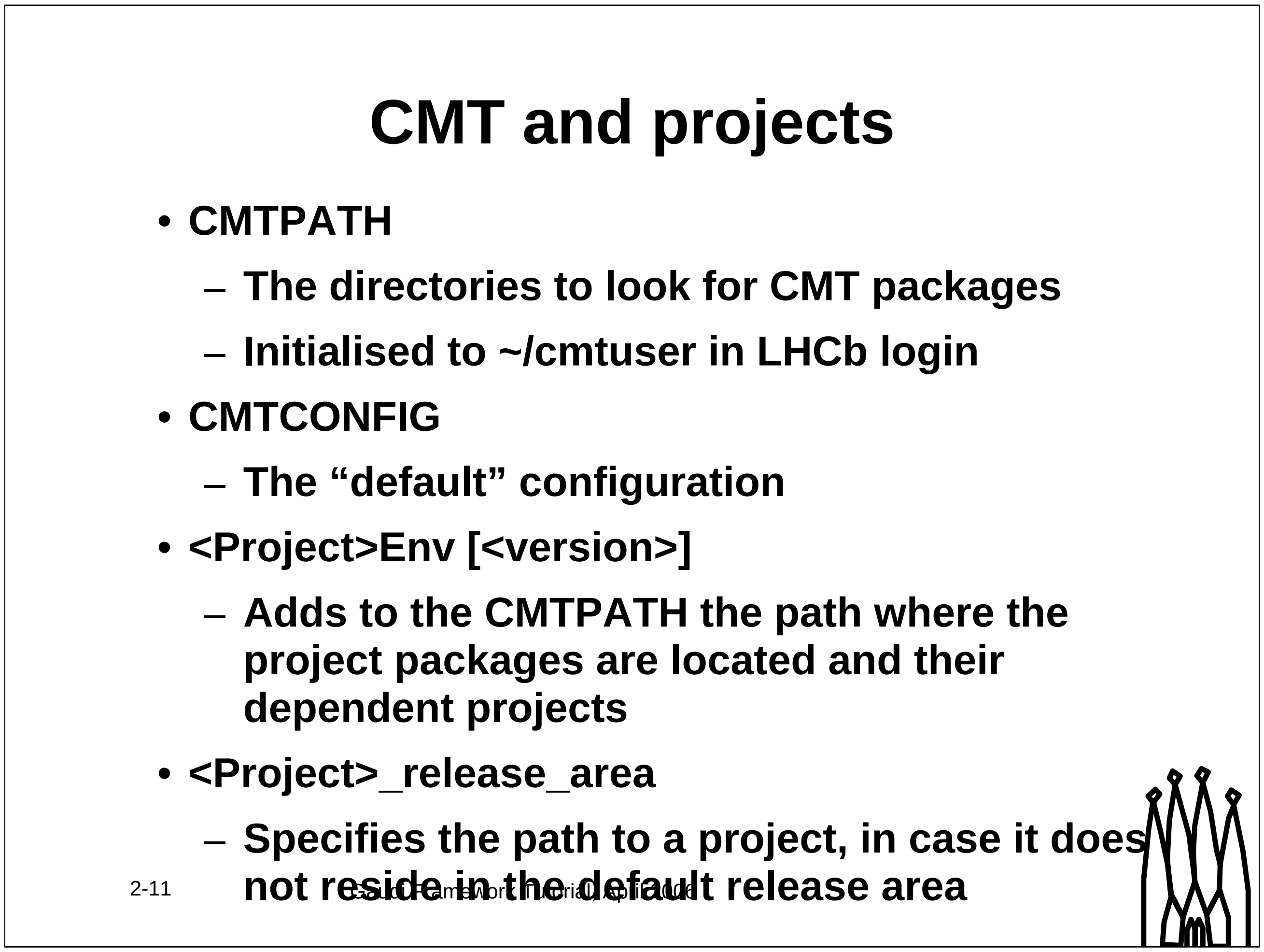
•**package**. Defines the name of the package

•**version**. Defines the version of the package. The number after the “v” is the *major* version number, that after the “r” the *minor* version number (“p” is also available, for *patches*). We increase the major version if there are major changes in functionality that are likely to affect other packages (e.g. modified interfaces, incompatible changes in data objects, changed behaviour of algorithms), the minor version otherwise (e.g. new classes added, backward compatible bug fixes)

•**use**. Instructs CMT on the dependencies of this package to other CMT packages. Normally the wildcard “*” is sufficient because the specific versions are frozen by the project. Explicit major versions can be a useful indication when integrating the software that the package can only work with a specific version of a dependent package. The name after the version is the location of the package (package group).
The “-no_auto_imports” token tells CMT that this package is needed at run time (so environment variables should be set up) but is not needed for compilation of linking (so compiler flags should not be imported)

•**application**. Tells CMT that this package is constituted of a program (application) and where to locate the sources. Similarly **library**

•**apply_pattern**. Tells CMT to apply one or more CMT commands (typically a **macro** definition) following a pattern that is common to all packages of a given type (in this case applications). Other common patterns are ***component_library*** and ***linker_library***.



## CMT and projects

- **CMTPATH**
  - **The directories to look for CMT packages**
  - **Initialised to ~/cmtuser in LHCb login**
- **CMTCONFIG**
  - **The “default” configuration**
- **<Project>Env [<version>]**
  - **Adds to the CMTPATH the path where the project packages are located and their dependent projects**
- **<Project>_release_area**
  - **Specifies the path to a project, in case it does not reside in the default release area**

2-11 Gaudi Framework Tutorial, April 2006

**CMT and projects**

•CMTPATH
Is the list of top directories to look for CMT packages. In LHCb environment is set at login time to:
/afs/cern.ch/user/<u>/<user>/cmtuser

•CMTCONFIG
Is the default configuration used by the CMT commands (when the –tag= option is not specified) . It is set by default at login time to the current platform name and recommended default compiler (e.g. slc3_ia32_gcc323). The User can change it to a different default (e.g. slc3_ia32_gcc323_dbg to turn on the debug configuration)

•<Project>Env script adds to this path the locations of the specific project environment (and dependent projects) with which you choose to work. For example, “LHCbEnv v20r3” sets CMTPATH to:

/afs/cern.ch/user/<u>/<user>/cmtuser:/afs/cern.ch/lhcb/software/releases/LHCB/LHCB_v20r3:/afs/cern.ch/lhcb/software/releases/DBASE:/afs/cern.ch/lhcb/software/releases/PARAM:/afs/cern.ch/sw/Gaudi/releases/GAUDI/GAUDI_v18r3:/afs/cern.ch/sw/lcg/app/releases/LCGCMT/LCGCMT_42b

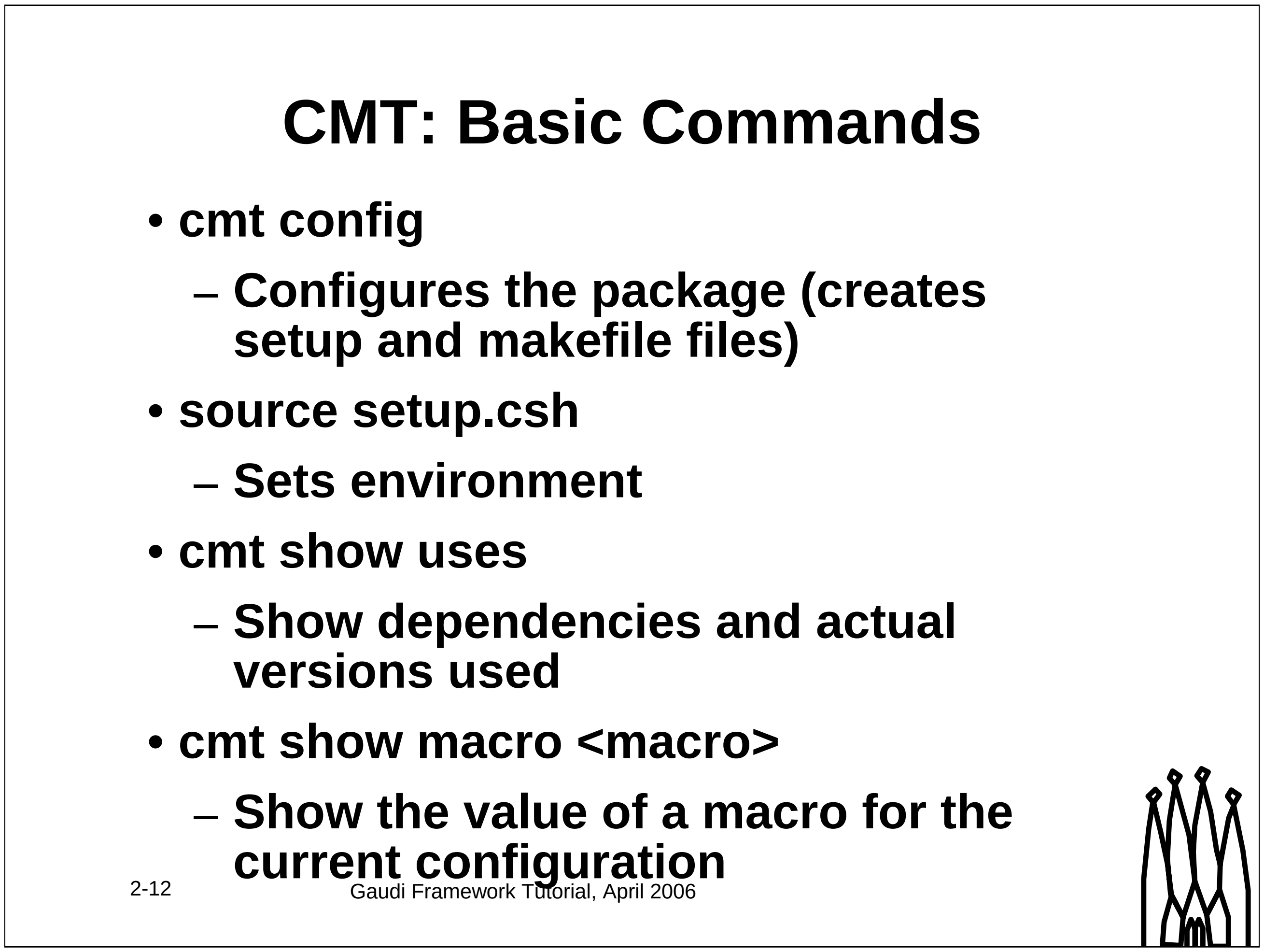
If given without argument, the script prints a list of all available versions

•<Project>_release_area is the location of project releases. For LHCb software it defaults to
$LHCb_release_area = /afs/cern.ch/lhcb/software/releases

The CMTPATH can be modified to point to a different area. E.g.
setenv LHCb_release_area $LHCBDEV
LHCbEnv v20r3

sets CMTPATH to:

/afs/cern.ch/user/<u>/<user>/cmtuser:/afs/cern.ch/lhcb/software/releases/**DEV**/LHCB/LHCB_v20r3:/afs/cern.ch/lhcb/software/releases/**DEV**/DBASE:/afs/cern.ch/lhcb/software/releases/**DEV**/PARAM:/afs/cern.ch/sw/Gaudi/releases/GAUDI/GAUDI_v18r3:/afs/cern.ch/sw/lcg/app/releases/LCGCMT/LCGCMT_42b



## CMT: Basic Commands

- cmt config
  - Configures the package (creates setup and makefile files)
- source setup.csh
  - Sets environment
- cmt show uses
  - Show dependencies and actual versions used
- cmt show macro <macro>
  - Show the value of a macro for the current configuration

2-12 Gaudi Framework Tutorial, April 2006

**CMT Primary commands**

•configure a package

```
> cd cmtuser/package/v1r0/cmt
> cmt config
> source setup.csh
```

•visualize packages and version numbers used by a library or an application.

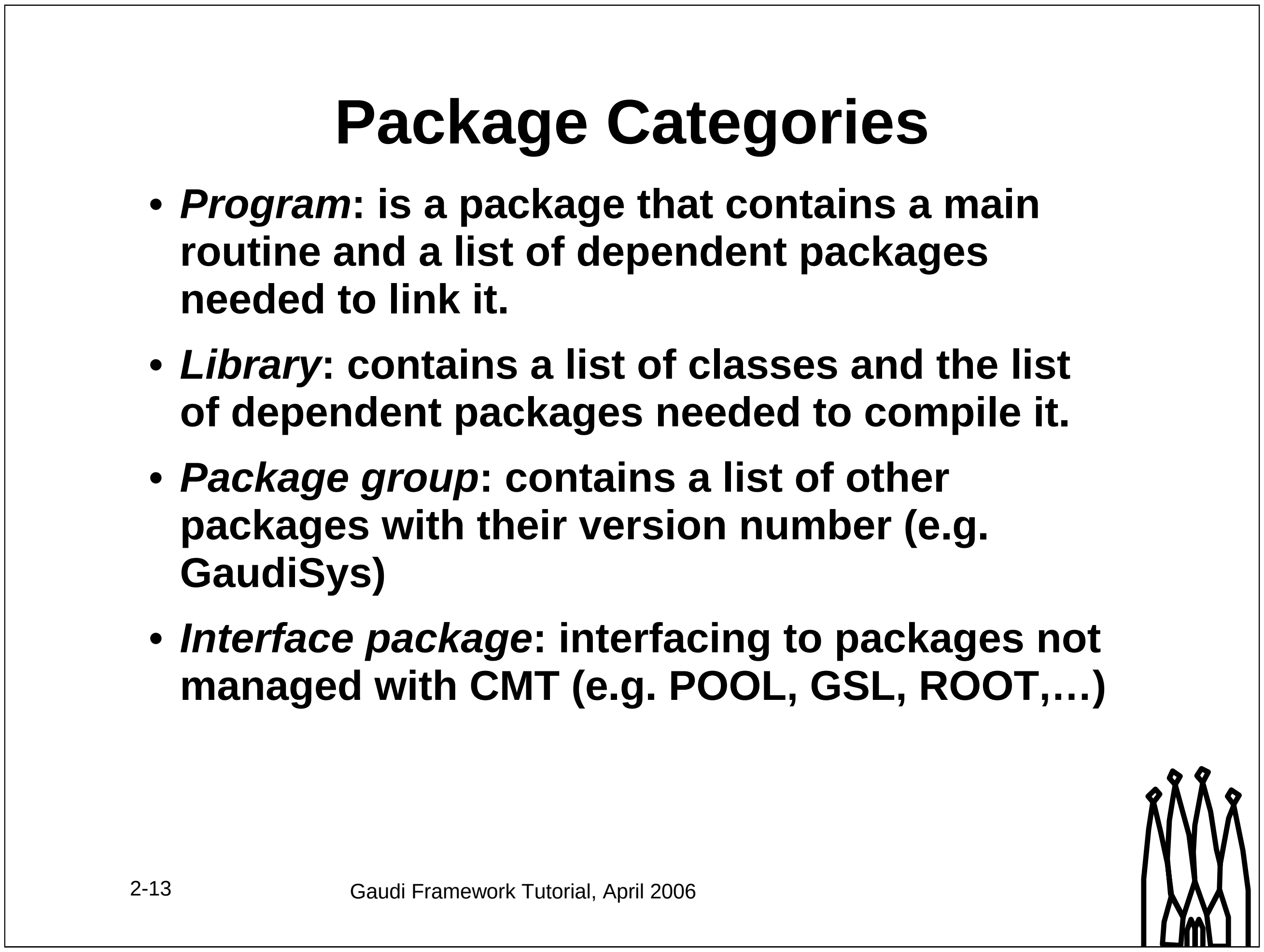
```
> cd cmtuser/package/v1r0/cmt
> cmt show uses
```

•get macro definitions used in makefiles

```
> cd cmtuser/package/v1r0/cmt
> cmt show macros
```

•get one specific macro used in makefiles

```
> cd cmtuser/package/v1r0/cmt
> cmt show macro cppflags
```



## Package Categories

- ***Program***: **is a package that contains a main routine and a list of dependent packages needed to link it.**
- ***Library***: **contains a list of classes and the list of dependent packages needed to compile it.**
- ***Package group***: **contains a list of other packages with their version number (e.g. GaudiSys)**
- ***Interface package***: **interfacing to packages not managed with CMT (e.g. POOL, GSL, ROOT,…)**

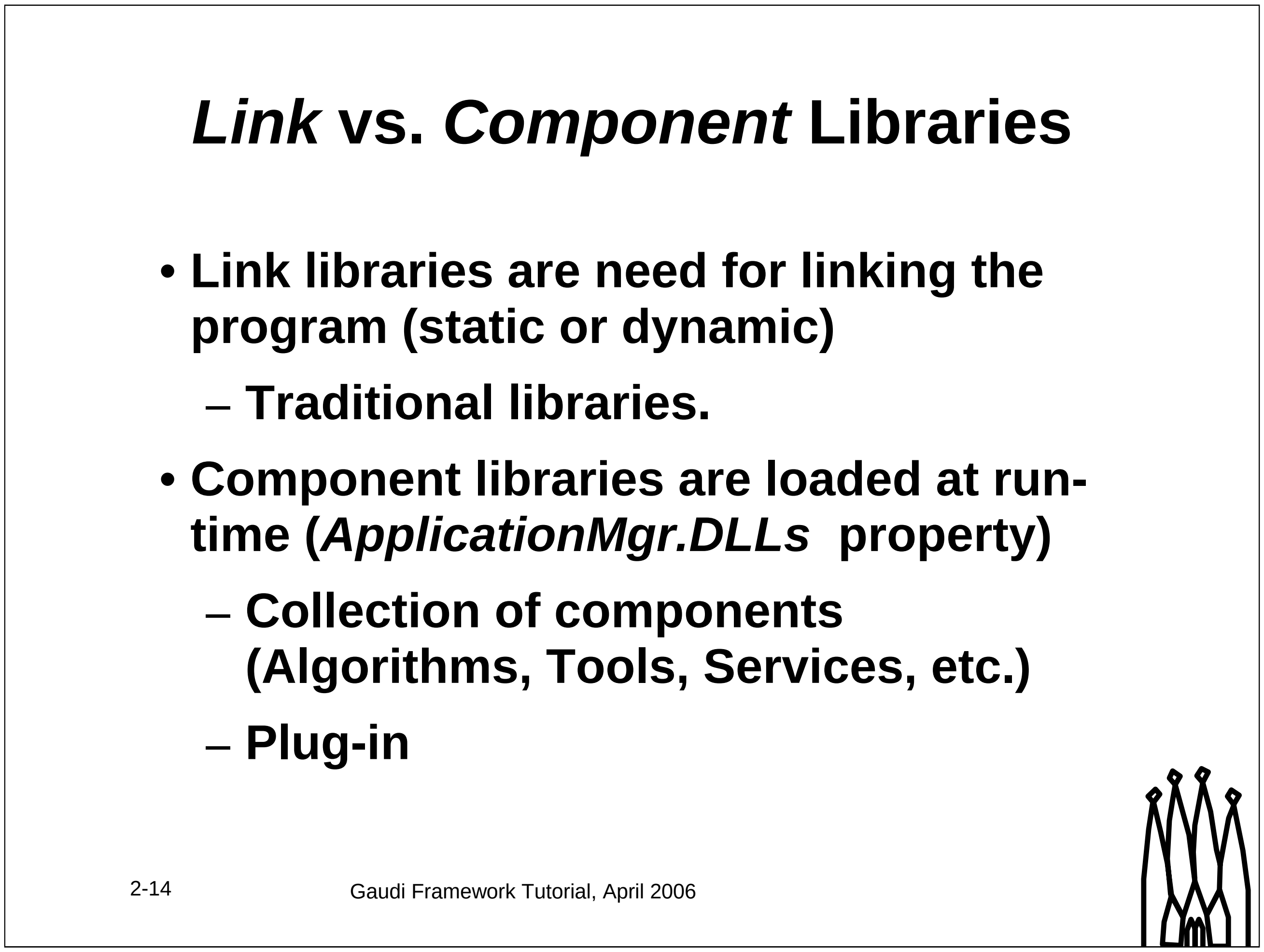
2-13 Gaudi Framework Tutorial, April 2006

**Package Categories**

With respect to CMT it is interesting to distinguish the different categories of packages. They are used in the exactly the same way but their requirement files will show some differences, specially in patterns that are used.

The concept of **interface package** is interesting for integrating in the build system packages that have been developed outside CMT. Basically these packages only define a number of macros and environment variables needed for compiling, linking and running with these external packages. These interfaces packages are defined currently in the LCGCMT project (common between the various LHC experiments using CMT)

**Package group** are useful for fixing a set on compatible versions of other packages. In this case the package using this set of packages needs only to state the version number of the package group.



## *Link* vs. *Component* Libraries

- Link libraries are need for linking the program (static or dynamic)
  - Traditional libraries.
- Component libraries are loaded at run-time (*ApplicationMgr.DLLs* property)
  - Collection of components (Algorithms, Tools, Services, etc.)
  - Plug-in

2-14 Gaudi Framework Tutorial, April 2006

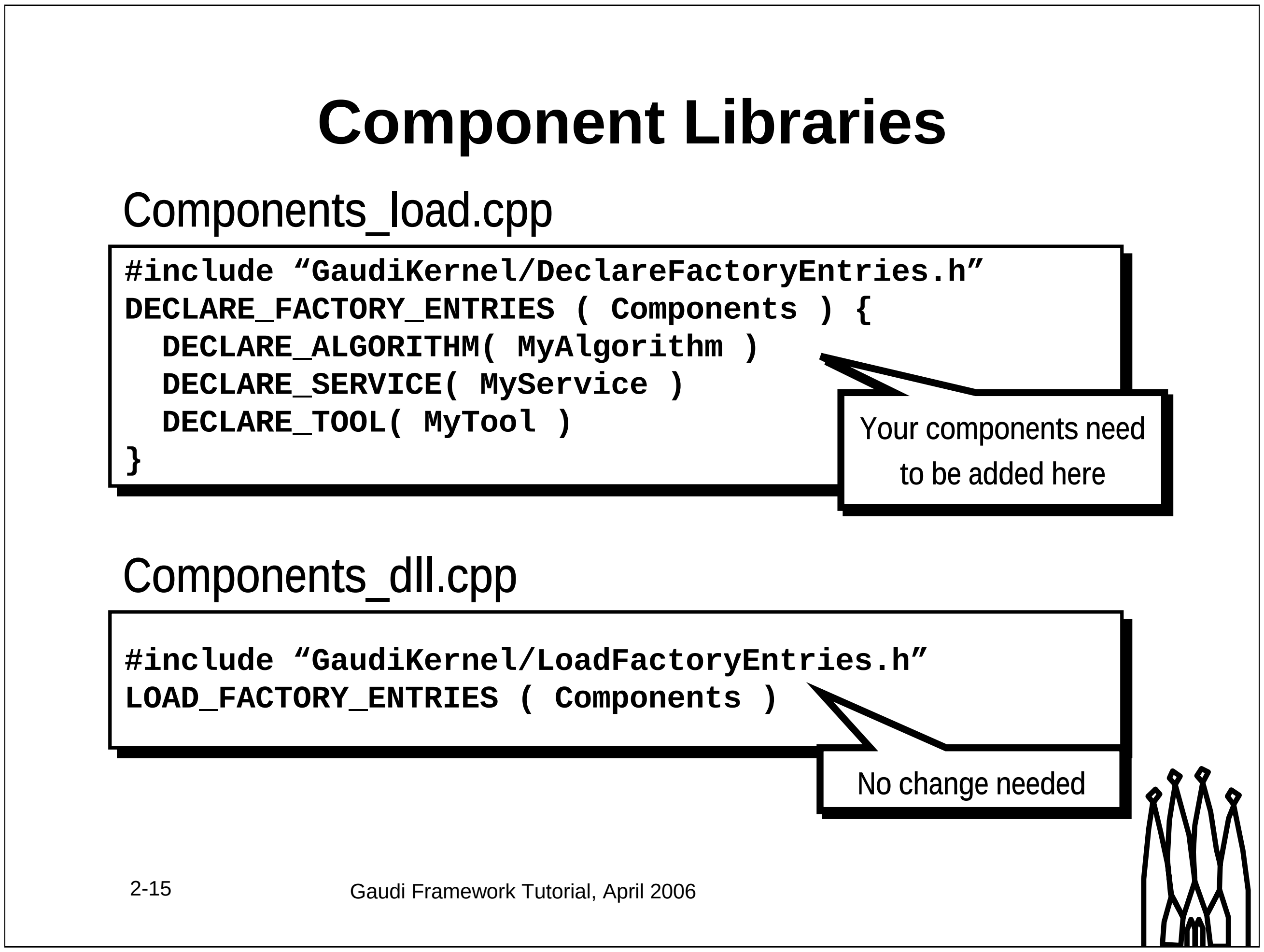
**Link libraries**

These are traditional libraries that, as the name implies, will be linked into the application. Typically they contain base classes or data objects. Remember that, if you rebuild a link library, you must also relink the application

**Component Libraries**

Component libraries are shared libraries that contain standard framework components which implement abstract interfaces. Such components are Algorithms, Auditors, Services, Tools and Converters. These libraries do not export their symbols apart from the one which is used by the framework to discover what components are contained in the library. These libraries are loaded at run time – this means that you do not have to relink the application if you rebuild the library

The Tutorial will be based on the development of a “component” library that will include all the Algorithms that we are going to develop during the practical exercises.



# Component Libraries

Components_load.cpp

```
#include "GaudiKernel/DeclareFactoryEntries.h"
DECLARE_FACTORY_ENTRIES ( Components ) {
  DECLARE_ALGORITHM( MyAlgorithm )
  DECLARE_SERVICE( MyService )
  DECLARE_TOOL( MyTool )
}
```

Your components need to be added here

Components_dll.cpp

```
#include "GaudiKernel/LoadFactoryEntries.h"
LOAD_FACTORY_ENTRIES ( Components )
```

No change needed

2-15 Gaudi Framework Tutorial, April 2006

## Component Libraries

In order to satisfy the requirements of a component library, two additional files must also be present in the package. One is used to declare the components, the other to load them. Because of the technical limitations inherent in the use of shared libraries, it is important that these two files remain separate, and that no attempt is made to combine their contents into a single file.

**<Components>_load.cpp** This file contain the declaration of all the components that should be available in the component library. There have been same macros defined to ease the writing of this file.

**<Components>_dll.cpp** This file is fixed and needs to be added into the package and do not need to be updated if new components are added later into the package.

## Getting a package

- The “getpack” command
  - Script combining “cvs checkout” + “cmt config”
  - It suggests the latest version of package

```
> getpack [hat/]<package> [<version>] [head]
```

2-16 Gaudi Framework Tutorial, April 2006

### Getting a package

The “getpack” command has been developed to ease the use for getting a copy of a package from the LHCb CVS repository and putting it in the correct directory structure with the correct version. It also suggests to the user what versions are available for the package in case the user does not specify it.

For the tutorial we recommend to get always the “head” revision of the packages.

## Building a package

- Working in the /cmt directory
  - <package>/<version>/cmt
- Invoke the gmake command

```
> gmake [target] [tag=<configuration>] [clean]

        configurations:   $CMTCONFIG (default)
                          $CMTDEB (for debug)
```

- Set the run time environment
  - (Not needed for building)

```
> source setup.csh [-tag=<configuration>]
```

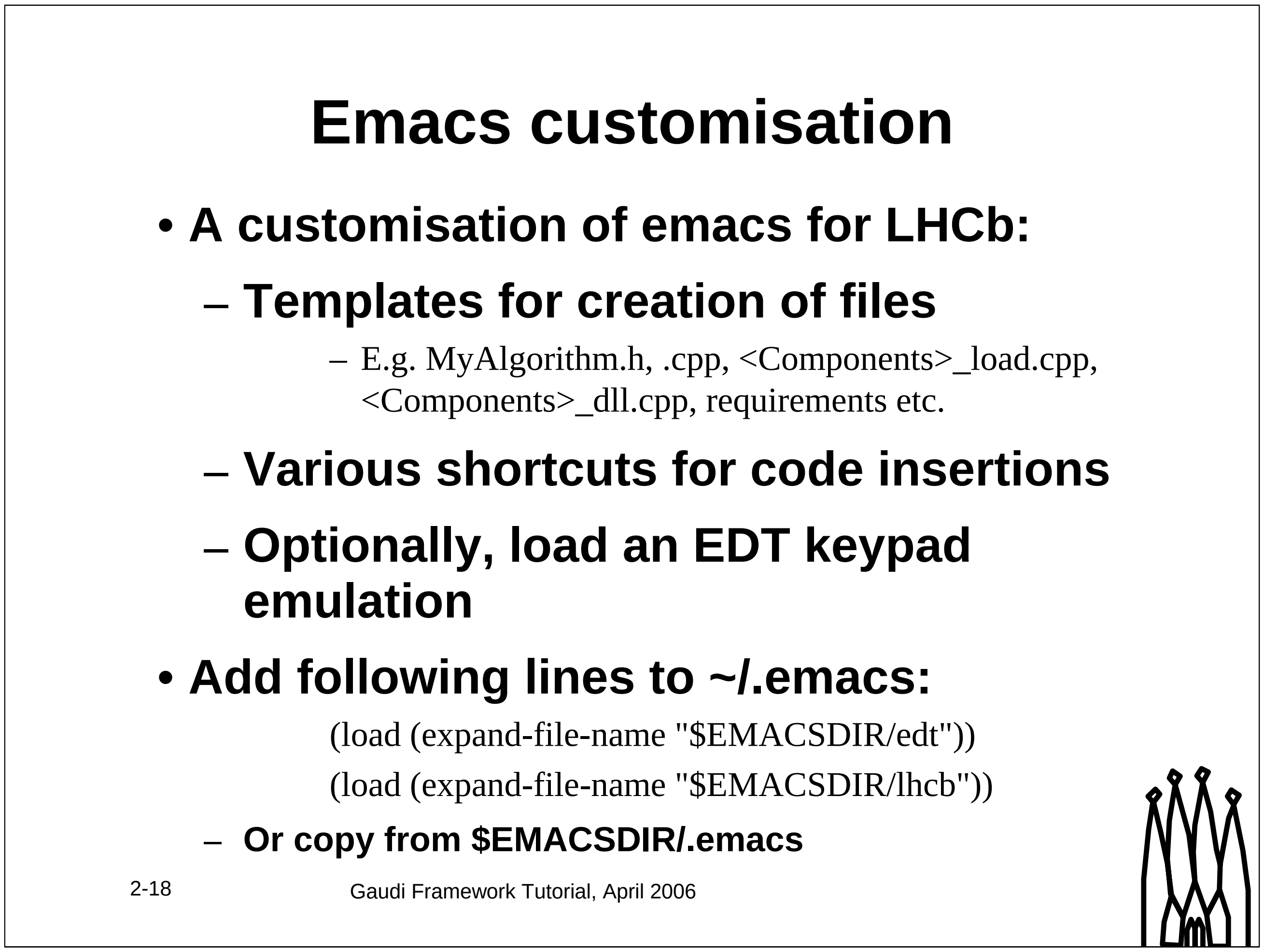
2-17 Gaudi Framework Tutorial, April 2006

**Building a package**

Typically for building (and also running) a package we stay in the /cmt directory. This is convenient for executing the configuration and build commands.

•**Setting the environment**. We need to setup the correct environment for the current version of the package. This environment consists on a set of environment variables (PATH, LD_LIBRARY_PATH, and others). Setting the environment is done my executing “source setup.csh” in the .cmt directory. This must be done before running the application always and in some cases also is needed for building the package. Therefore, we suggest to do it before building the package. You only need to re-do the setting of the environment if you change the package or the version you are using otherwise the environment stays valid for the complete session. We suggest for the tutorial to use the “debug” configuration by issuing the command “source setup.csh –tag=$CMTDEB”

•**The gmake command**. The “gmake” command is used for building the libraries and/or the programs. There are various configurations available in form of “tag=<configuration>”. We suggest for the tutorial to use the “debug” configuration by issuing always the command “gmake tag=$CMTDEB” since the default is without debug information. Note that “gmake” and “make” are equivalent on lxplus



## Emacs customisation

- A customisation of emacs for LHCb:
  - Templates for creation of files
    - E.g. MyAlgorithm.h, .cpp, <Components>_load.cpp, <Components>_dll.cpp, requirements etc.
  - Various shortcuts for code insertions
  - Optionally, load an EDT keypad emulation
- Add following lines to ~/.emacs:

```
(load (expand-file-name "$EMACSDIR/edt"))
(load (expand-file-name "$EMACSDIR/lhcb"))
```

  - Or copy from $EMACSDIR/.emacs

**LHCb emacs customisation**

Documentation at:
http://agenda.cern.ch/askArchive.php?a01680/a01680s6t2/transparencies/EmacsConfig.pdf

## Tutorial Packages

- **Tutorial/Main [v5r3]**
  - **The *executable*. During the tutorial will be using the same program.**
- **Tutorial/Components [v7r0]**
  - **A package consisting of a single *Component* library in which we will be adding all the Algorithms of the Tutorial**
- **Tutorial/TutKernel [v1r0]**
  - **A package containing *public Headers*, used in the later part of the tutorial**

**Tutorial/Main**

This is the main program. We will use it without modification during the tutorial.

| | |
|---|---|
| /cmt/requirements | requirements file |
| /options/jobOptions.opts | job options file to be used during the tutorial (everything is commented to start with) |

**Tutorial/Components**

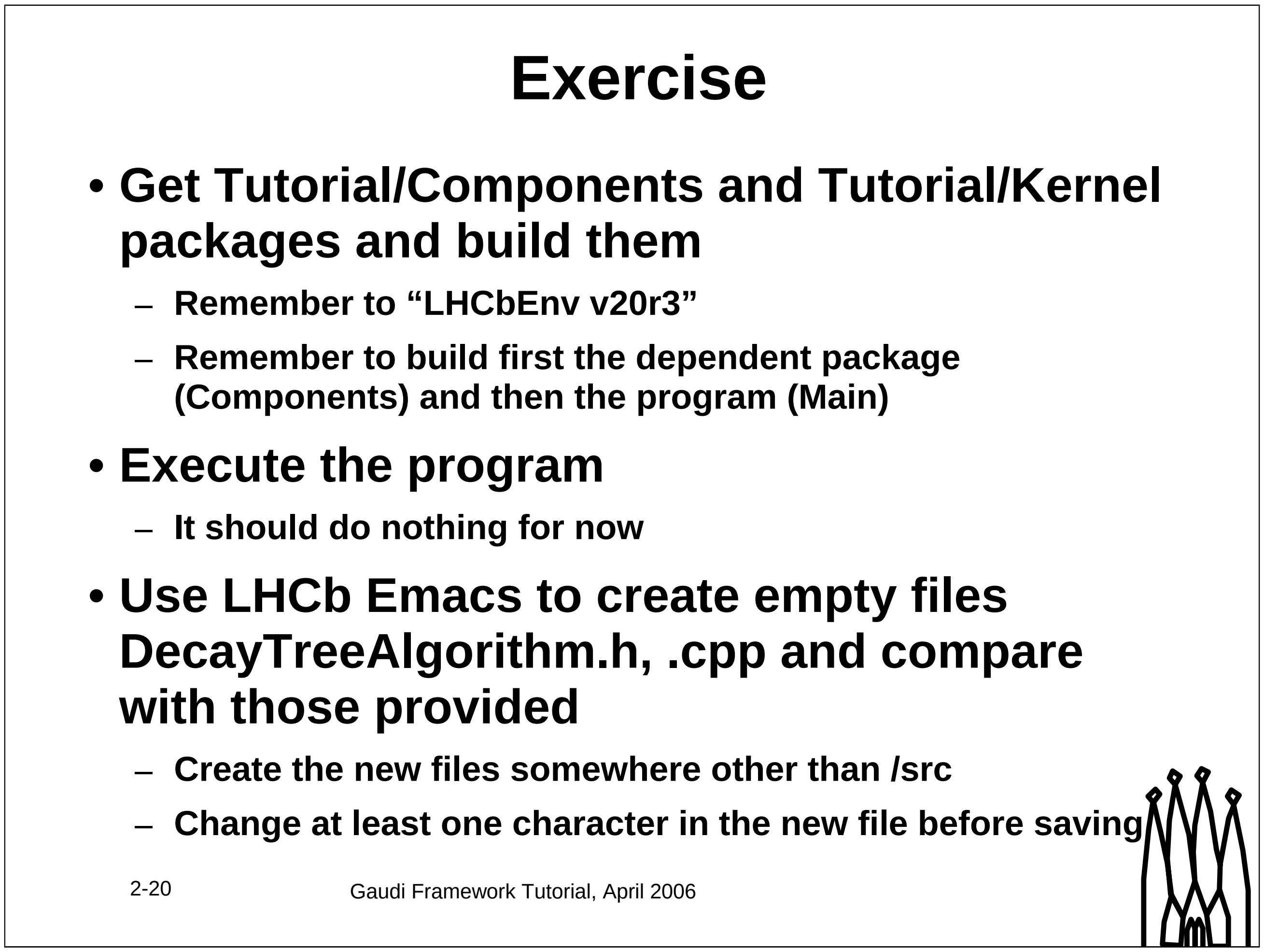
This is the component library. We will populate the /src directory with new files from other /src.<xxxx> which contains the solutions to the different exercises of the tutorial.

| | |
|---|---|
| /cmt/requirements | requirements file |
| /options/<exercise>.opts | job options “fragments” for the different exercises |
| /src/Component_dll.cpp | Needed for building a “component library” |
| /src/Component_load.cpp | Needed for building a “component library” |
| /src/DecayTreeAlgorithm.* | “Empty” .h and .cpp files for the Decay Tree exercise |
| /src.decaytree/* | Directory with solutions for the Decay Tree exercise |
| /src.hist_tuple/* | Directory with solutions for the Histogram Ntuple exercise |
| /src.data/* | Directory with solutions for the Writing Data exercise |
| /src.usetool/* | Directory with solutions for the Tools exercise |
| /src.solution/* | Directory with the complete solution |

**Tutorial/TutKernel**

Contains Event classes and tool interfaces. We will use these in the last two exercises

| | |
|---|---|
| /xml/VertexInfo.xml | File for generation of LHCb event model class used in Writing Data exercise |
| /Kernel/IMCAcceptanceTool.h | Interface for GaudiTool of Tools exercise. |



## Exercise

- Get Tutorial/Components and Tutorial/Kernel packages and build them
  - Remember to “LHCbEnv v20r3”
  - Remember to build first the dependent package (Components) and then the program (Main)
- Execute the program
  - It should do nothing for now
- Use LHCb Emacs to create empty files DecayTreeAlgorithm.h, .cpp and compare with those provided
  - Create the new files somewhere other than /src
  - Change at least one character in the new file before saving

2-20 Gaudi Framework Tutorial, April 2006

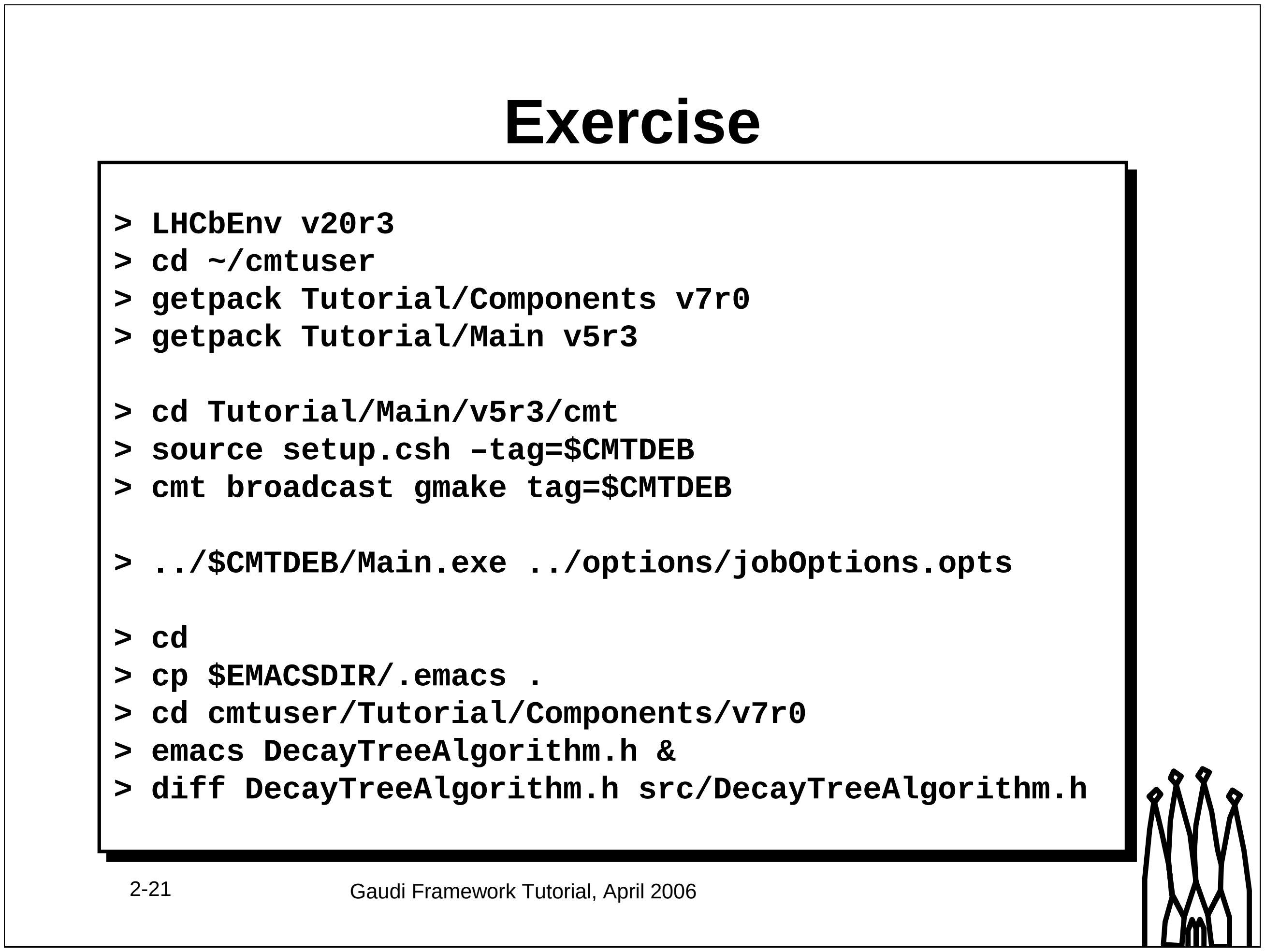
You can in fact build both packages in one go:

go to the cmt directory of Tutorial/Main package (i.e. the application)

type:

```
cmt broadcast make
```

This command tells cmt to execute the “make” command for all this package and any dependent packages that are found on the first part of the CMTPATH (i.e. ~/cmtuser)



## Exercise

```
> LHCbEnv v20r3
> cd ~/cmtuser
> getpack Tutorial/Components v7r0
> getpack Tutorial/Main v5r3

> cd Tutorial/Main/v5r3/cmt
> source setup.csh –tag=$CMTDEB
> cmt broadcast gmake tag=$CMTDEB

> ../$CMTDEB/Main.exe ../options/jobOptions.opts

> cd
> cp $EMACSDIR/.emacs .
> cd cmtuser/Tutorial/Components/v7r0
> emacs DecayTreeAlgorithm.h &
> diff DecayTreeAlgorithm.h src/DecayTreeAlgorithm.h
```

2-21 Gaudi Framework Tutorial, April 2006

Note that the file automatically generated by emacs must be touched before saving it (e.g. add a comment) – otherwise emacs will consider that you have not edited anything and will not save the template.