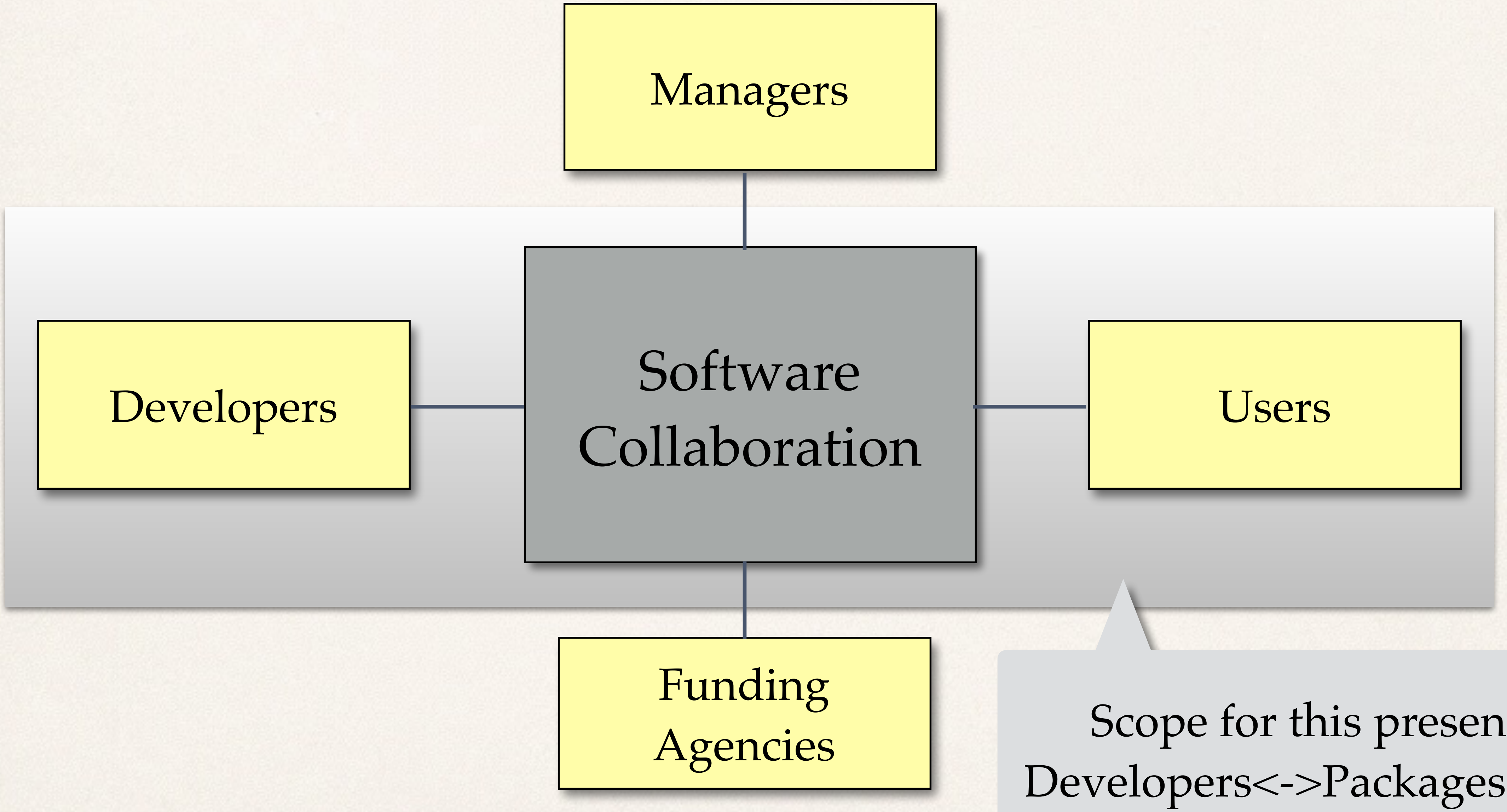
Model for Organising and Supporting Development Activities

Pere Mato/CERN

HEP Software Collaboration Meeting, 3 April 2014

Collaboration Stakeholders



Managers

Developers

Software Collaboration

Users

Funding Agencies

Scope for this presentation
Developers<->Packages<->Users

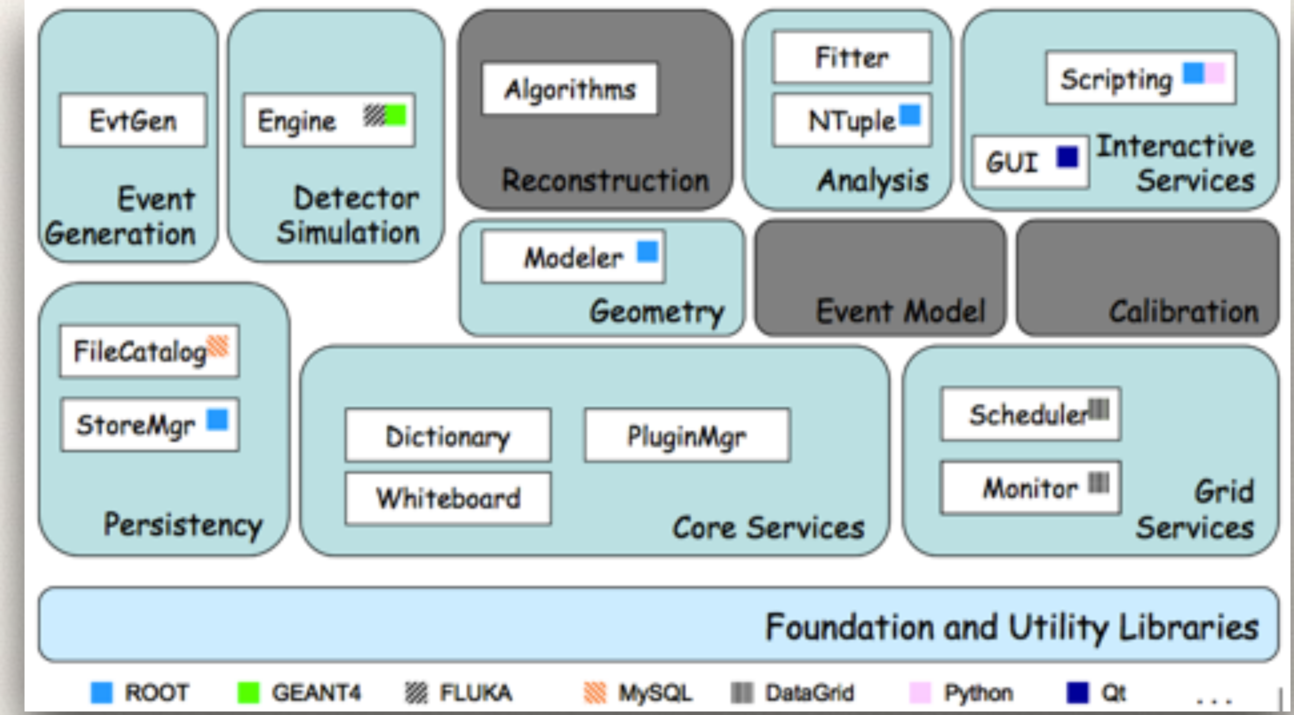
Outline

- The Software
    - Structure, domains, catalogs
    - Software Interoperability
    - Modularity and package dependencies
- The Developers
    - Project independence
    - Development aids and software process
- The Users
    - Lowering usage barriers
    - Feedback

The Software

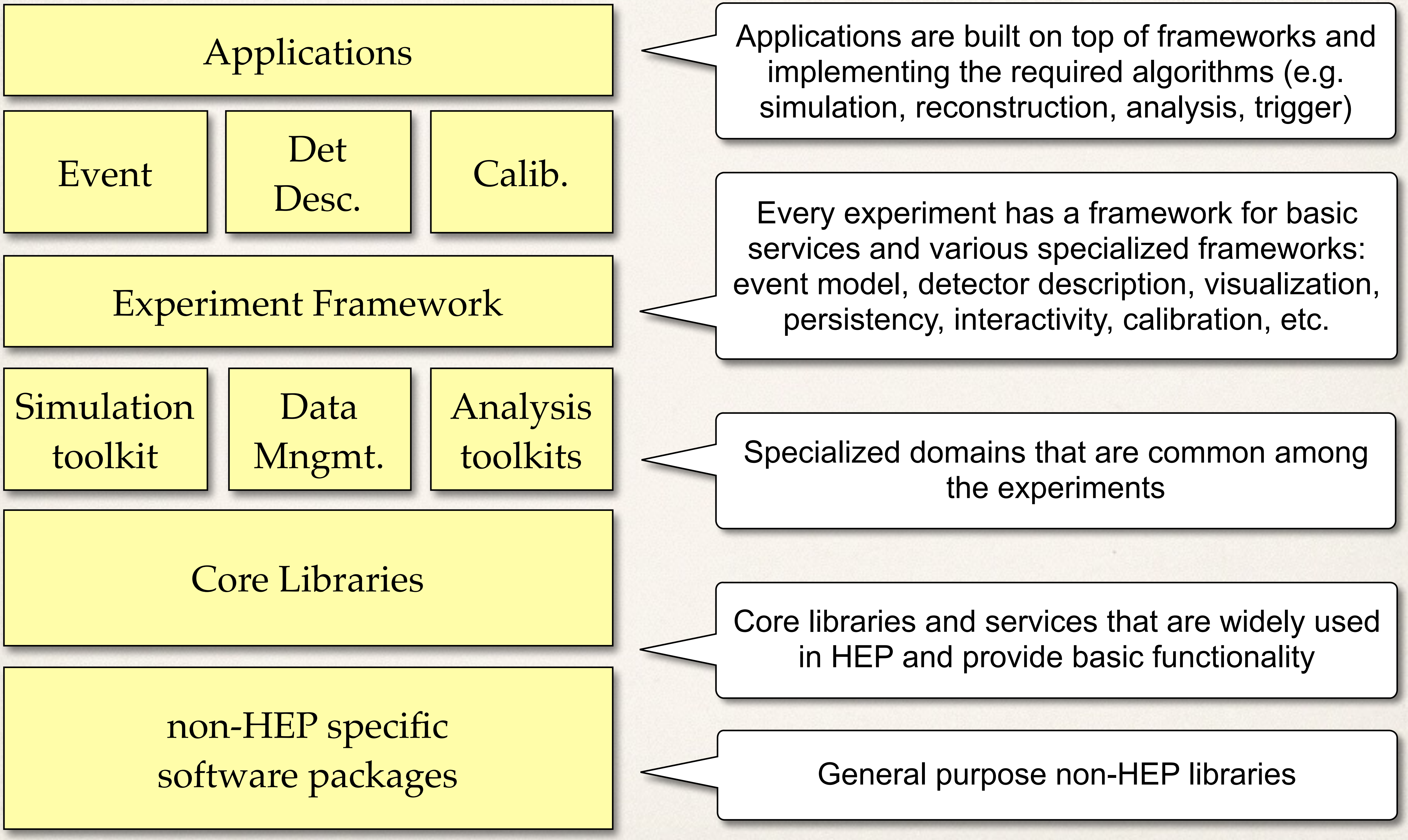
A Sea of Software Packages

- The Collaboration goal is to **develop** software components [packages] in collaboration and make them available to the HEP [scientific] community **to use** them
  - Facilitate package development
  - Facilitate package usage
- We expect different sort of packages of different **levels** (foundation, core, generic, specialized) in different software **domains** (simulation, statistics, math, graphics, etc.)
- Nothing very new so far (same ideas 12 years ago)
  - A good moment to re-think about modularity



Domain decomposition from *LHC Computing Grid Project Architecture Blueprint RTAG*, 2002

Typical HEP Software Stack



Applications

Event

Det Desc.

Calib.

Experiment Framework

Simulation toolkit

Data Mngmt.

Analysis toolkits

Core Libraries

non-HEP specific software packages

Applications are built on top of frameworks and implementing the required algorithms (e.g. simulation, reconstruction, analysis, trigger)

Every experiment has a framework for basic services and various specialized frameworks: event model, detector description, visualization, persistency, interactivity, calibration, etc.

Specialized domains that are common among the experiments

Core libraries and services that are widely used in HEP and provide basic functionality

General purpose non-HEP libraries

Software Components

- Foundation Libraries
  - Basic types
  - Utility libraries
  - System isolation libraries
- Mathematical Libraries
  - Special functions
  - Minimization, Random Numbers
- Data Organization
  - Event Data
  - Event Metadata (Event collections)
  - Detector Conditions Data
- Data Management Tools
  - Object Persistency
  - Data Distribution and Replication

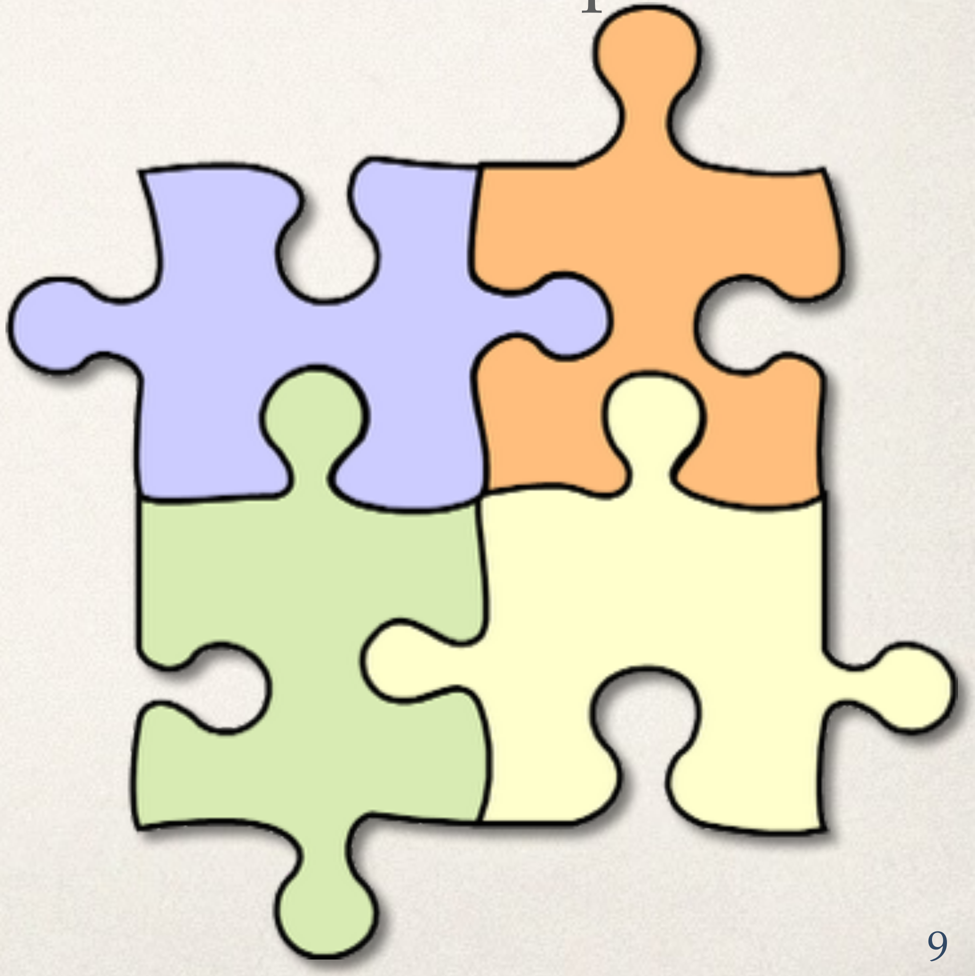
Simulation Toolkits
  Event generators
  Detector simulation
Statistical Analysis Tools
  Histograms, N-tuples
  Fitting
Interactivity and User Interfaces
  GUI
  Scripting
  Interactive analysis
Data Visualization and Graphics
  Event and Geometry displays
Distributed Applications
  Parallel processing
  Grid computing

AIDA WP2 Example

- The common software work package of EU AIDA project is delivering a set of generic software toolkits for geometry and reconstruction (9 partners)
  - Some of them developed in the context of one experiment but abstracted and packaged in experiment-independent manner
- Some of the packages are:
  - USolids - Unified 3D shapes library
  - DD4hep - Toolkit for describing detectors
  - aidaTT - Tracking toolkit
  - PandoraPFA - particle flow algorithms
  - tkLayout - track trigger simulation
  - Bach - telescope reconstruction and alignment
  - Arbor - Topological clustering

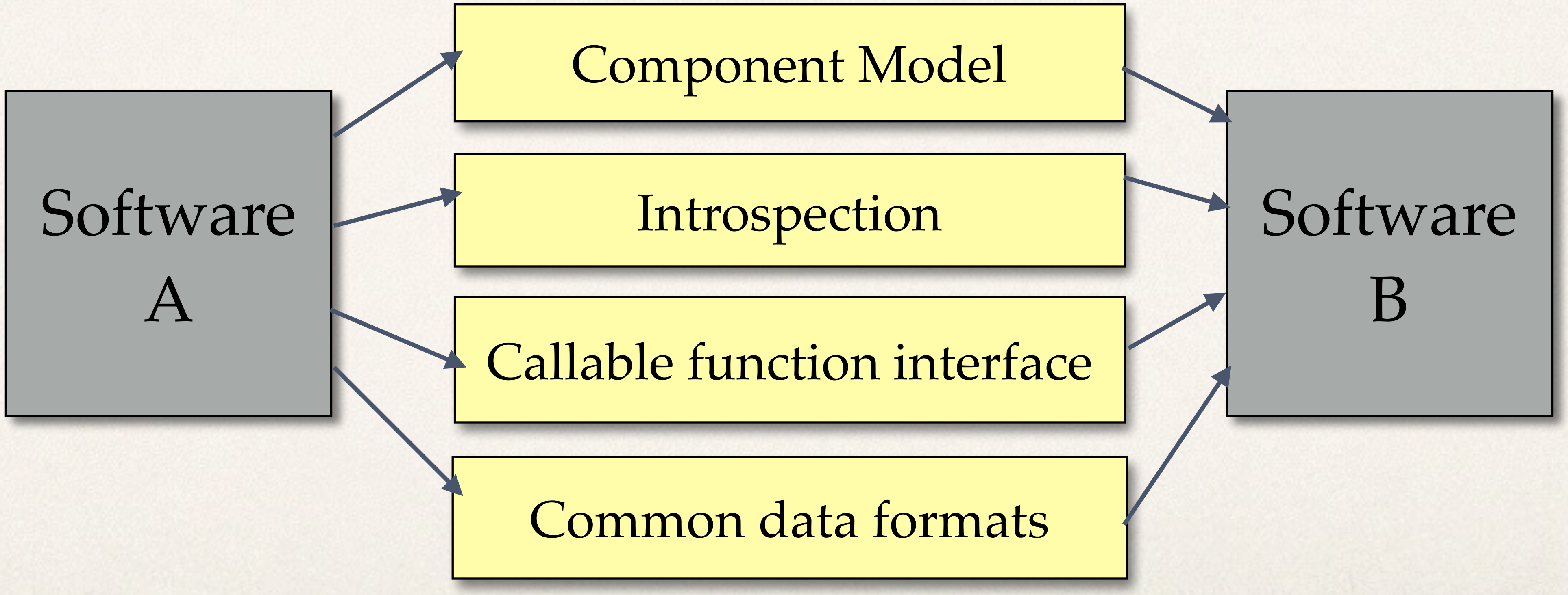
Interoperability

- Capability of different software components to exchange data via a common set of exchange formats or interfaces
- Software components need to interoperate with other components to provide the required functionality
    - There is not software package that is standalone and complete
    - No even ROOT :-)
- There are several levels of interoperability
    - From very loosely coupled to strongly coupled



Software Interoperability Levels

- Level 0 - Common Data Formats
- Level 1 - Callable Interface
- Level 2 - Introspection Capabilities (generic callable interface)
- Level 3 - Component Model (common framework)



Interoperability Levels (1)

- Level-0: Common Data Formats
  - Allows interoperation between different programming languages, different hardware, etc.
  - Examples: HepMC Event Record, LCIO event model, GDML
- Level-1: Callable Interface
  - The basic calling interface provided by any programming language
  - It implies to agree on the language (e.g. C++), on the version of the standard (e.g. c++11), on version of the compiler (e.g. gcc 4.8), etc.
  - Other "details" to take into account: exceptions throwing and handling; const-ness and thread-safety; side effects; dependencies to other libraries; runtime environments
  - Examples: many libraries such as fastjet, CLHEP, Boost, etc.

Interoperability Levels (2)

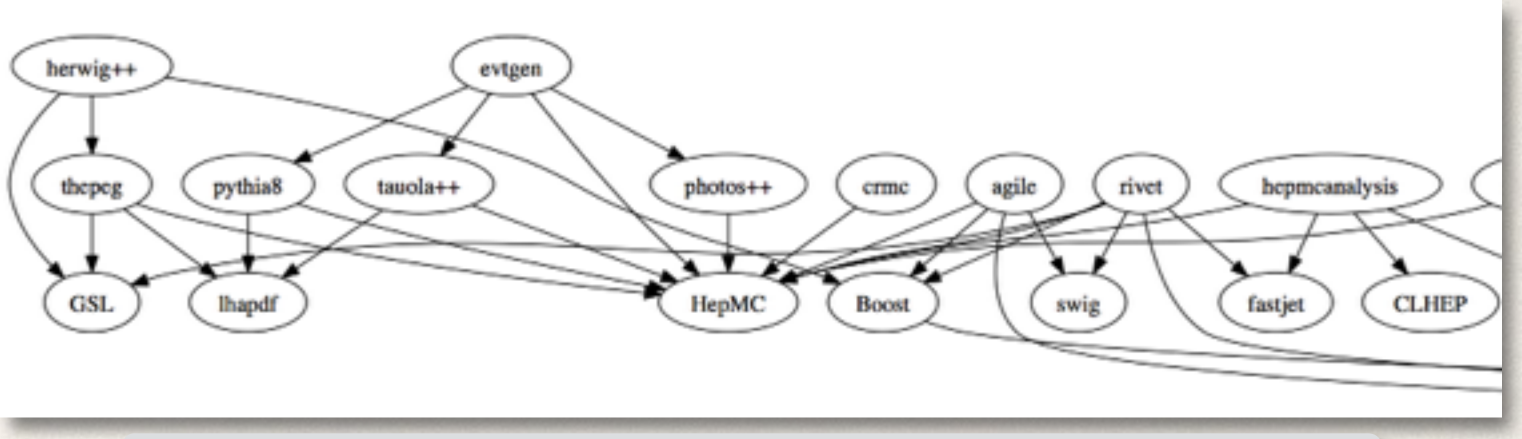
- Level 2 - Introspection Capabilities
  - Software elements to facilitate the interaction of objects in a generic manner such as **Dictionaries** and **Scripting** interfaces
  - Example: PyROOT, which is a Python extension module that allows the user to interact with any ROOT (C++) class from the Python interpreter
- Level 3 - Component Model
  - Software components of a “common framework” offers maximum re-use
  - ‘standard’ way to configure its parameters, to log and report errors, manage object lifetime and ownership rules, ‘standard’ plug-in management, etc.
  - Unfortunately, not a single Framework has been generally adopted

Interoperability: Take Away Messages

- Software packages will interoperate with other packages at different levels
    - Forcing the integration of all packages done at the same level is a mistake
- We will need to define data exchange formats, APIs, component models, etc.
- Some questions will need to be answered for each package
    - What basic functionality it requires?
    - Can be re-used on different frameworks?
    - Complexity and specificity of the exchanged data?
    - etc.

Package Dependencies

- Very few packages are truly standalone (not having any dependency)
  - Very often packages depend of other packages
- Package dependencies are difficult to manage
  - Complicates the configuration, the build process, the distribution and the deployment
- Avoiding dependencies is not a good solution in general
  - Adds code duplication
  - Reduces code re-use
- **Managing dependencies is essential**
  - ‘Standards’ and tools are required



Fragment of the dependency graph of some MC generator packages

Defining Configurations

- We will need to define ‘working’ configurations with all the packages taking into account their dependencies and version constrains

```
# Application Area Projects
LCG_AA_project(COOL  COOL_2_8_17)
LCG_AA_project(CORAL CORAL_2_3_26)
LCG_AA_project(RELAX RELAX_1_3_0k)
LCG_AA_project(ROOT  5.34.05)
LCG_AA_project(LCGCMT LCGCMT_${heptools_version})

# Externals
LCG_external_package(4suite            1.0.2p1                                  )
LCG_external_package(AIDA              3.2.1                                    )
LCG_external_package(blas              20110419                                 )
LCG_external_package(Boost             1.50.0                                   )

...

# Generators
LCG_external_package(starlight         r43            MCGenerators/starlight    )
LCG_external_package(herwig            6.520          MCGenerators/herwig       )
LCG_external_package(herwig            6.520.2        MCGenerators/herwig       )
LCG_external_package(crmc              v3400          MCGenerators/crmc         )
LCG_external_package(cython            0.19           MCGenerators/cython       )
LCG_external_package(yaml_cpp          0.3.0          MCGenerators/yaml_cpp     )
```

For example for the LCG external software, a single file lists all the packages and their required versions to define a complete configuration

Packaging and Distribution

- Tar-balls are great, but difficult (up to impossible) to upgrade, uninstall, and simply keep track of them
- We will need to develop or adopt some easy-to-use system for compiling, installing, and upgrading HEP software
  - Examples: MacPorts, Fink, APT, etc.
  - Multi-platform (Unix, Mac, Win) support is required
- Perhaps even better to distribute the software using CernVM-FS
  - All HEP software will be 'virtually' installed automatically

Key to the success of the HEP Software Collaboration

Turn-Key Systems

- In addition to individual software packages providing a specific functionality there is the need to provide **complete turn-key systems**
- Small experiments or experimental programs cannot afford to build custom solutions (from pieces) for their data processing applications
  - For example the experiment studies of FCC
- We need to provide complete solutions including
  - Data processing framework, event data model, detector description
  - Built-in generators and detector simulation
  - Basic reconstruction and analysis tools
  - etc.

The Developers

Contributing to the Repository

- Development teams will **collaborate to populate the repository** with common software packages of different software domains
  - Ideally only one package for a given functionality
- Packages to be included in the repository will need to conform to some standards
  - e.g. required documentation, build procedures, dependencies declaration, version naming convention, test definitions, etc.
- There probably should be a ‘submission’ procedure
  - Steps the developer goes through to get a package accepted by the Collaboration
  - Including a more or less formal review process

Lifecycle

- Software libraries or packages lose their value over time if not maintained
- The Collaboration should have a central ‘**issue tracker**’ from which to alert developers of potential maintenance needs
    - Issues (bugs) can be dispatched from the central one to the specific developer one
- Developers should plan to maintain their library over time
    - If the developer no longer can or wish to maintain the package, it may became an ‘orphan’ package
- The Collaboration should have a mechanism to assign ‘orphan’ packages to other maintainers
    - Perhaps this is a role for the larger HEP laboratories

Project Independency

- Each development team should **keep its autonomy**
    - The Collaboration should not enforce any particular software process, project management or methodology
    - Each team may use its own repository, bug tracker, web project site, user forum, etc.
    - But the Collaboration may provide them if needed
- ‘Ownership’ of the package resides with its developers
    - A clear way of recognition and proper credits
    - At the same time the developers need to ensure support and maintenance

Software Process and Tools

- The Collaboration should **not enforce a given software process** but it should encourage a common set of 'best practices' and tools
  - Creating a kind of consensus would certainly facilitate integration, testing, distribution and deployment
- Build Systems
  - For example the HEP community is using more and more CMake
- Testing
  - Defining and running tests should be strait-forward
  - CMake companion tool, CTest could be a good proposal
- Documentation
  - Reference documentation from comments in code (i.e. Doxygen)
  - User guides

Continuous Integration and Testing

- In general the package must portable and not restricted to a particular compiler or operating system
    - Testing of the portability should be ensured by the Collaboration by building and running tests on many platforms
- Interoperability between packages should also be validated
    - Complete builds of all packages for a number of configurations should be done by the Collaboration
- A continuous build and testing system should be setup with clear reports to developers

No file changed as of **Tuesday, April 01 2014 - 03:00 CEST**

Experimental

Site	Build Name	Update	Configure		Build		Test			Build Time
		Files	Error	Warn	Error	Warn	Not Run	Fail	Pass	
ec-slc6-x86-64-spi-8	x86_64-slc6-gcc48-opt	0	0	0	0	0	0	5	84	11 hours ago
ec-slc6-i686-spi-1	i686-slc6-gcc48-opt	0	0	0	0	0	0	4	65	11 hours ago
ec-slc6-x86-64-spi-9	x86_64-slc6-icc14-dbg	0	0	0	19	1	0	43	26	9 hours ago

Preview

Site	Build Name	Update	Configure		Build		Test			Build Time
		Files	Error	Warn	Error	Warn	Not Run	Fail	Pass	
macitois17.cern.ch	x86_64-mac108-gcc42-opt	0	0	0	10	1	0	9	38	10 hours ago
lxbsp0516.cern.ch	x86_64-slc5-gcc43-opt	0	0	0	0	0	0	4	65	11 hours ago
ec-slc6-x86-64-spi-9	x86_64-slc6-gcc48-opt	0	0	0	0	0	0	5	84	11 hours ago
ec-slc6-x86-64-spi-8	x86_64-slc6-icc14-dbg	0	0	0	19	1	0	38	31	8 hours ago

Release

Site	Build Name	Update	Configure		Build		Test			Build Time
		Files	Error	Warn	Error	Warn	Not Run	Fail	Pass	
macitois17.cern.ch	x86_64-mac108-clang34-opt	0	0	0	16	1	0	11	37	8 hours ago
macitois18.cern.ch	x86_64-mac108-gcc42-opt	0	0	0	16	1	0	11	37	11 hours ago
ec-slc6-x86-64-spi-7	x86_64-slc6-gcc48-opt	0	0	0	0	0	0	6	86	11 hours ago
lxbsp0516.cern.ch	x86_64-slc5-gcc43-opt	0	0	0	0	0	0	4	67	11 hours ago

Added Value for Developers

- Channel for advertising quality software to a large scientific community
  - Make good packages known to potential users
- Ensuring the coherency with the packages from other colleagues
  - This facilitate their integration into systems
- Providing integration builds and integration tests
  - Validation on a extended set of platforms (architectures, OS and compilers)
- Providing distribution repositories to the community
  - Easy to locate, select and install the required package

The Users

Lowering the Barriers for Re-use

- Cataloging all packages and tools to avoid the re-invent syndrome
  - If people know what exists, perhaps they will re-invent less
- Users should get what they need very easily and no more
  - ROOT is a very good example of easy installation, however the model has been “take all or nothing”
  - Similarly with Geant4, many specialized physics processes are built and installed by default and they are most of the time not needed
- We can do much better
  - For example in R, additional modules are downloaded, build and installed at runtime when required

- Good moment to **re-think modularity** requirements

Feedback

- The Collaboration needs the participation of their user communities to identify bugs in their software, as well as to plan for new features
  - It is essential to encourage user feedback
- A bug report is a gift to developers
  - As long as it contains all the required information and details to help the developer to reproduce the problem
- The Collaboration should organize the collection, triage and dispatching of all feedback issues and bugs

Added Value for Users

- All HEP packages in a single repository with standard documentation, build procedures, file structures, coherent interfaces, etc.
  - Even better, all pre-installed in the CernVM-FS system!
- Users will no need to re-code functionality that is available in repository
  - Assuming that the package is working, high quality, performant and well integrated to the rest of the software packages
- No fear to add the needed extra dependencies because building and installation will be complete automatic (transparent)
- Easy and centralized channel for providing feedback and bug reports

Summary

- Expect to populate a repository with a variety of software packages
  - Different levels (foundation, core, generic, specialized) and in different software domains
- These components will need to interoperate with other components to provide the required functionality
  - Definition of data formats, interfaces, API, component models required
- Standard software process should not be enforced
  - Encourage a common set of common practices and tools to facilitate integration and testing
- Each development team should keep its autonomy and way of working
  - Code repository, bug tracker, forum, web site, etc.
- Lowering the barriers for users to use any package in the repository
  - Easy-to-use installation system