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

Designing Architectures and Frameworks for HEP

Pere Mato (CERN)

Outline

- HEP [LHC] Data processing Overview
- Software project scale
- Architectural Design

    <break>

- Software Frameworks
- Example: GAUDI
- Software integrating elements

LHC computing characteristics

- Large number of physicicists and engineers participating actively in the data analysis and for extended period of time
- Widely distributed computing environment
- Huge quantity of data that has to be distributed and shared by all members of each experiment

Data Flow and Processing stages

LHC software requirements

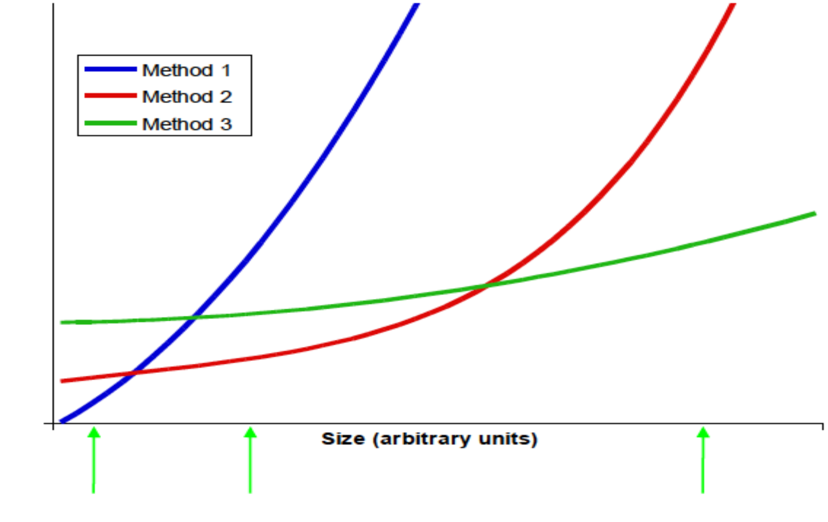
- Design should take into account the >15 years lifetime of the LHC
  - Resilient designs, technology choices will evolve over time
- The standard language for physics applications software in all four LHC experiments is C++
  - language choice may change in the future or multi-language could be introduced
- Operate seamlessly in a distributed environment and also be functional in ‘disconnected’ local environments
- Modularity of components with well-defined interfaces and interchangeability of implementations

LHC software requirements (2)

- Integrate well in a coherent software framework and other tools
- Favor software re-use. Use of existing software should be consistent with the architecture
- The software quality of the framework should be at least as good than the internal software of any of the sub-detectors
- Multi-Platforms. Software should be written in a portable manner and conformant to the language standards

Software Scale

- Small problems can be solved with simple techniques
- For large problems you need to use different techniques that are in general more complex and with a up front cost



Architecting a dog house

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

Architecting a house

- Built most efficiently and timely by a team
- Requires
  - Modeling
  - Well-defined process
  - Power tools

Architecting a high rise

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

Tools for large projects

- To make communication possible to you need to share a vocabulary
  - Standards for languages, design patterns, **architecture**, etc.
- To work together you need to control the integrity of source code
  - Tools for code versioning (e.g. CVS, SubVersion)
- To build, test and release a large system can be difficult
  - Tools for creating releases (e.g. CMT, SCRAM), tracking problems

→ But individual effort is still important
  - Good tools and methods can help to do a better job

Performance

- "*More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity*", William Wulf (AT&T Professor)
- Overall efficiency is what really matters
  - The cost of the improving the code (people are expensive) should be included
- Perceived performance is what really matters
  - Is the system getting the job done or not?
- Reminder: Performance assumes correctness
  - A fast program delivering [sometimes] wrong results is not helpful

Importance of Reuse

- Put extra effort into building high quality components
- Be more efficient by re-using these components
- Many obstacles to overcome
  - too broad functionality / lack of flexibility in components
  - organisational – reuse requires a broad overview to ensure unified approach
    - we tend to split into domains each independently managed
  - cultural
    - don't trust others to deliver what we need
    - fear of dependency on others
    - fail to share information with others
    - developers fear loss of creativity

➔Reuse is a management activity

Application Domains

- Event Processing Applications
  - Trigger, Simulation, Reconstruction, Selection programs
- Data Analysis
  - Event/Detector display, data presentation programs
- Detector calibration
  - Calibration and Alignment programs
- Job configuration, submission, monitoring and control
  - Grid awareness

Mainly batch oriented. Interactive for development & debugging. Real-time.

Mainly interactive

Batch and interactive

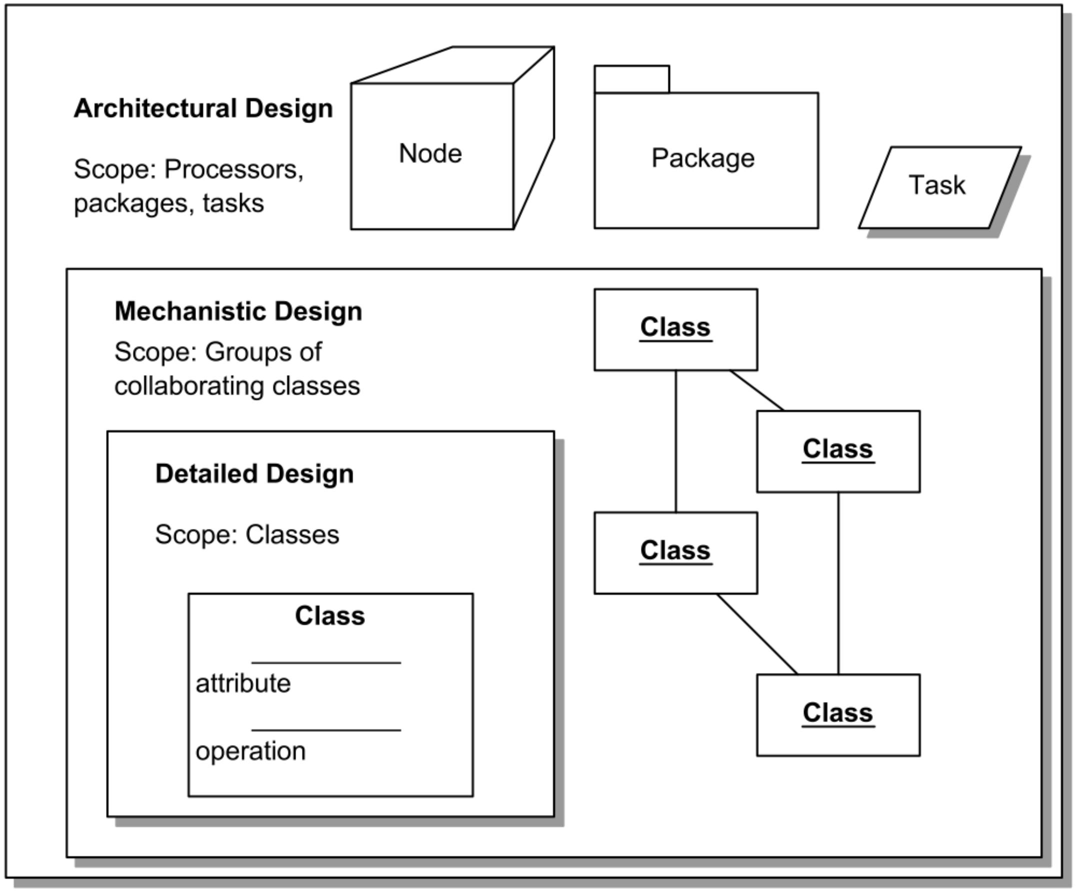
Mainly interactive

No Disjoint Domains

- For example, the LHCb requirements for interactive analysis:
  - Better [than PAW] integration with experiment framework
    - consistent with the analysis batch environment, use the same toolkits and experiment algorithms/tools
    - access the experiment data objects and allow browsing
    - integrated with event display
    - allow interactive reconstruction and simulation

Software Design

- System Architecture
- Component design
- Class design



Architectural Design

- Capture major interfaces between subsystems and packages early
- Be able to visualize and reason about the design in a common notation
  - Common vocabulary, running scenarios
- Be able to break the work into smaller pieces that can be developed concurrently by different teams
- Acquire an understanding of non-functional constrains
  - Programming languages, concurrency, database, GUI, component re-use

Architecture Defined

- Definition of [software] architecture [1]
  - Set or **significant decisions** about the **organization** of the software system
  - Selection of the **structural elements** and their **interfaces** which compose the system
  - Their **behavior** -- collaboration among the structural elements
  - **Composition** of these structural and behavioral elements into progressively larger **subsystems**
  - The **architectural style** that guides this organization
- The **architecture** is the blue-print (architecture description document)

[1] I. Jacobson, et al. “The Unified Software development Process”, Addison Wesley 1999

Mary Shaw, CMU
Grady Booch,
Philippe Kruchten,
Rich Reitman
Kurt Bittner, Rational

Architecture defined (continued)

- Software architecture also involves
  - usage
  - functionality
  - performance
  - resilience
  - reuse
  - comprehensibility
  - economic and technology constraints and tradeoffs
  - aesthetic concerns

Architectural Design Qualities

- A well designed architecture has certain qualities:
  - layered subsystems
  - low inter-subsystem coupling
  - robust, resilient and scalable
  - high degree of reusable components
  - clear interfaces
  - driven by most important and risky use cases
  - easy to understand

Models

- Models are the language of designer, in many disciplines
- Models are representations of the system to-be-built or as-built
- Models are vehicle for communications with various stakeholders
- Visual models, blueprints
- Scale
- Models allow reasoning about some characteristic of the real system

Many stakeholders, many views

- Architecture is many things to many different interested parties
  - end-user
  - customer
  - project manager
  - system engineer
  - developer
  - architect
  - maintainer
  - other developers
- Multidimensional reality
- Multiple stakeholders
  - → multiple views, multiple blueprints

Architectural design workflow

- Select scenarios: criticality and risk
- Identify main classes and their responsibility
- Distribute behavior on classes
- Structure in subsystems, layers, define interfaces
- Define distribution and concurrency
- Implement architectural prototype
- Derive tests from use cases
- Evaluate architecture
  - Iterate

Use case view

Logical view

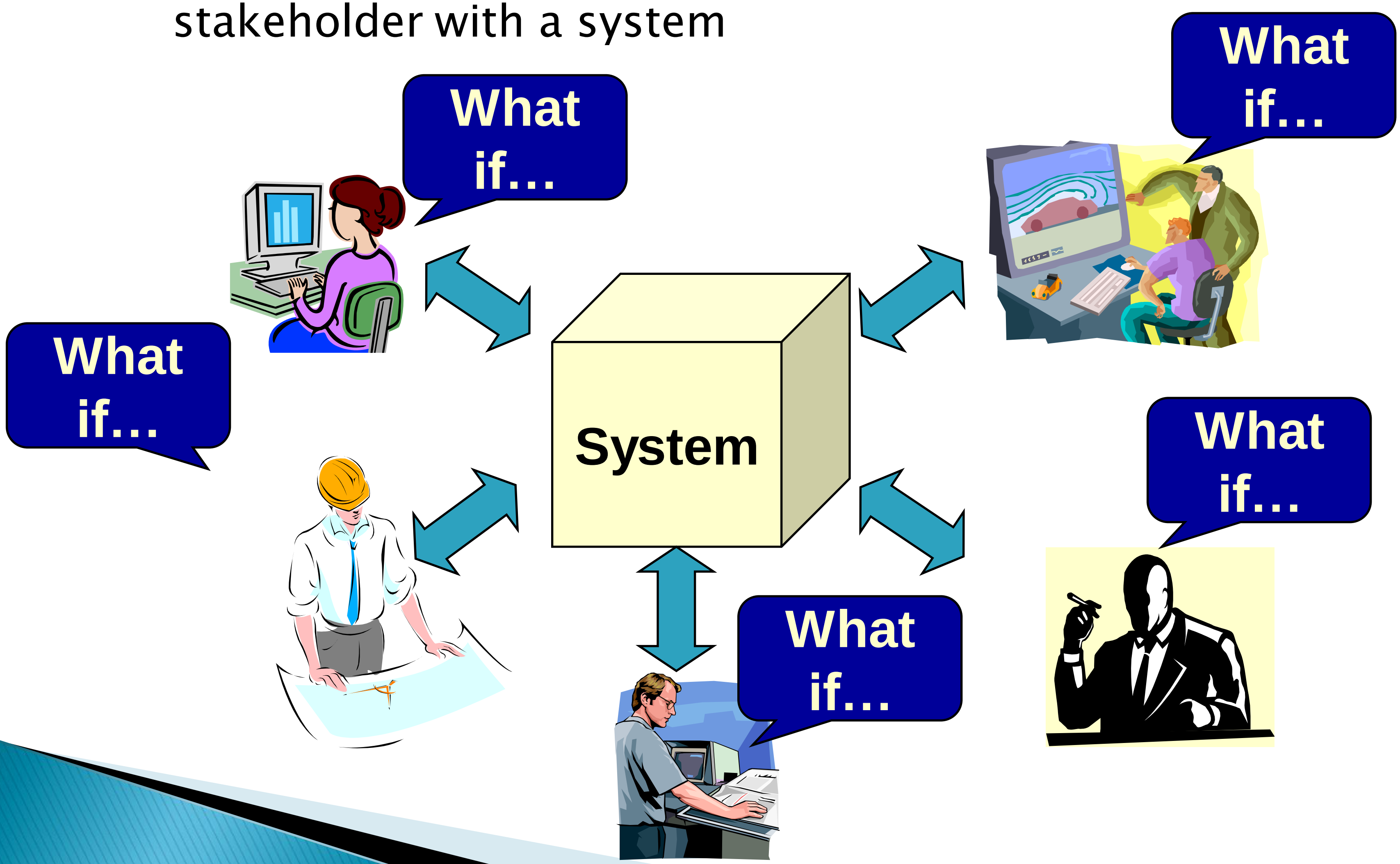
Implementation view

Deployment view

Process view

Scenario-based evaluation

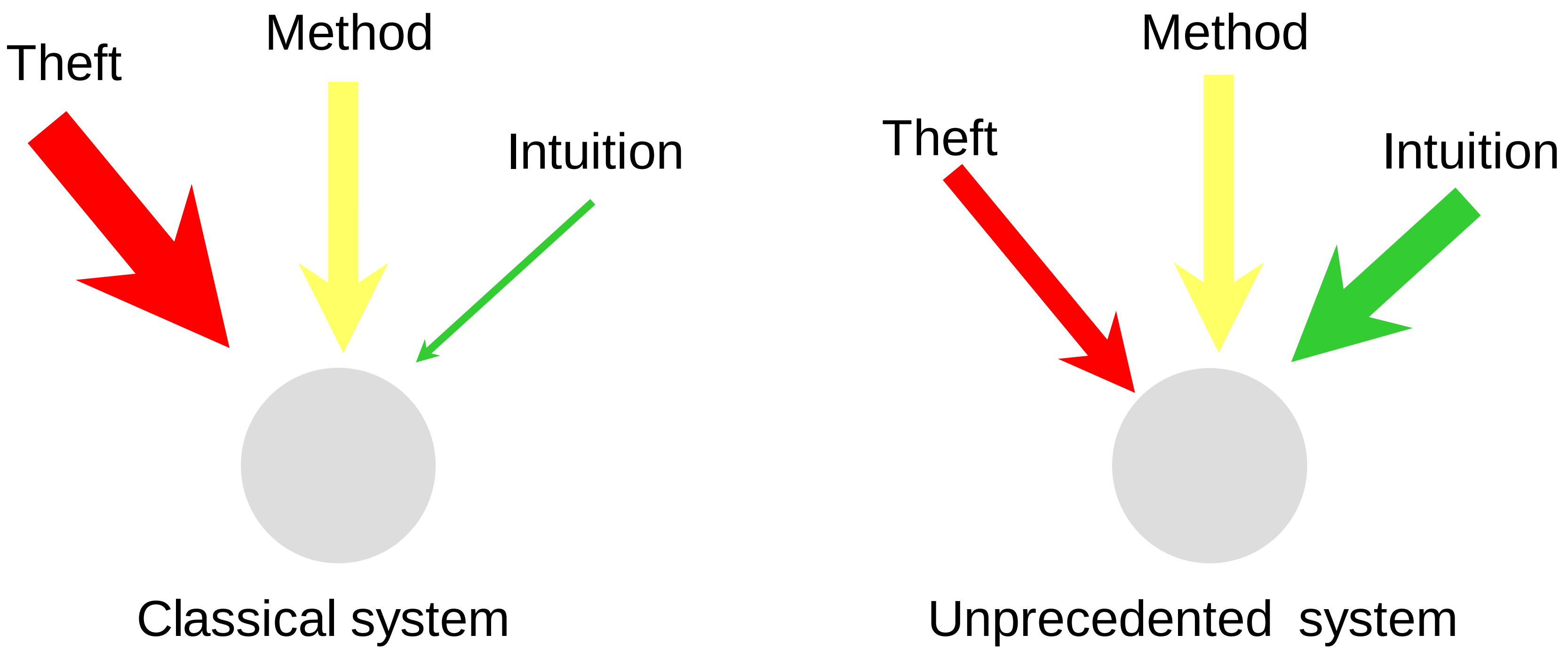
- Scenario is a brief description of an interaction of a stakeholder with a system



Scenarios evaluation examples

- User scenarios
  - What if I want to run a new track fit algorithm?
  - What if I need to use the newest calibration?
- Deployment engineer
  - What if we need to port the software to the Solaris platform?
  - What if we embed the software in real-time systems
- Manager
  - What if we need to support some standard data formats
  - What if we integrate a commercial GUI system

Sources of architecture



Architectural style

- An architecture style defines a family of systems in terms of a pattern of structural organization.
- An architectural style defines
  - a vocabulary of components and connector types
  - a set of constraints on how they can be combined
  - one or more semantic models that specify how a system's overall properties can be determined from the properties of its parts

Architectural styles

- General categorization of systems [1]

user-centric focus on the **direct visualization**
and manipulation of the objects
that define a certain domain

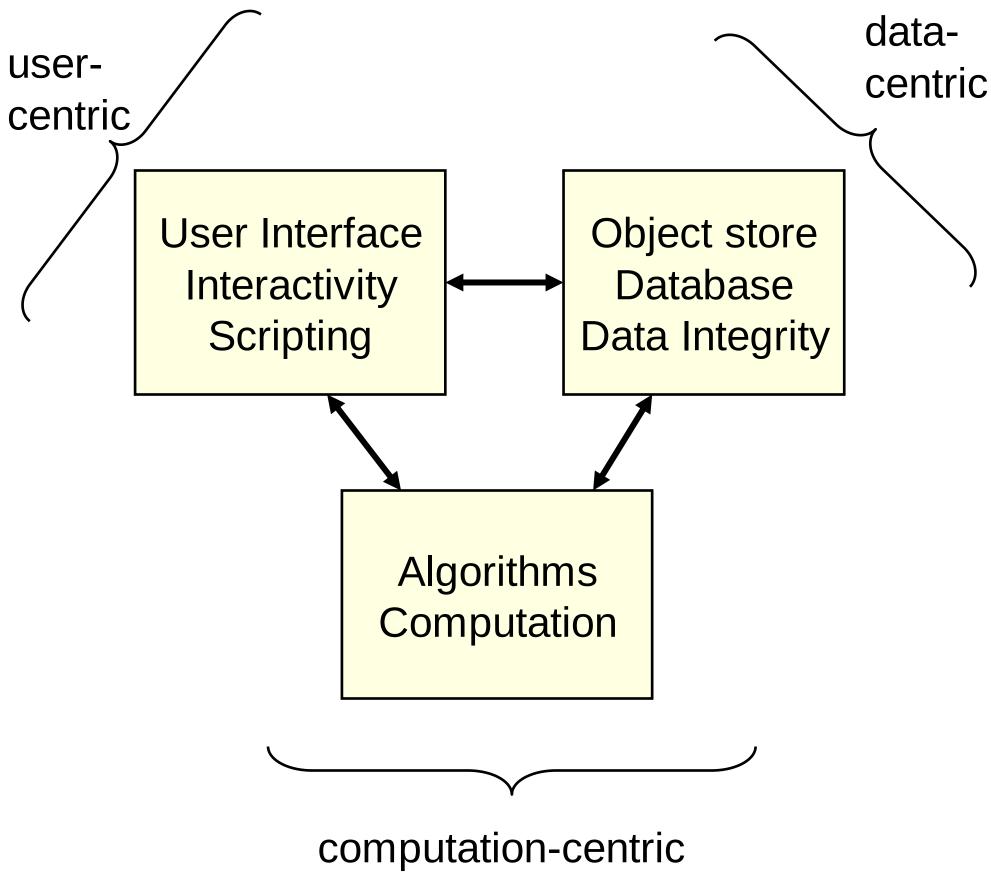
data-centric focus upon preserving the **integrity of the persistent**
objects in a system

computation-centric focus is on the **transformation**
of objects that are interesting
to the system

[1] G. Booch, "Object Solutions", Addison-Wesley 1996

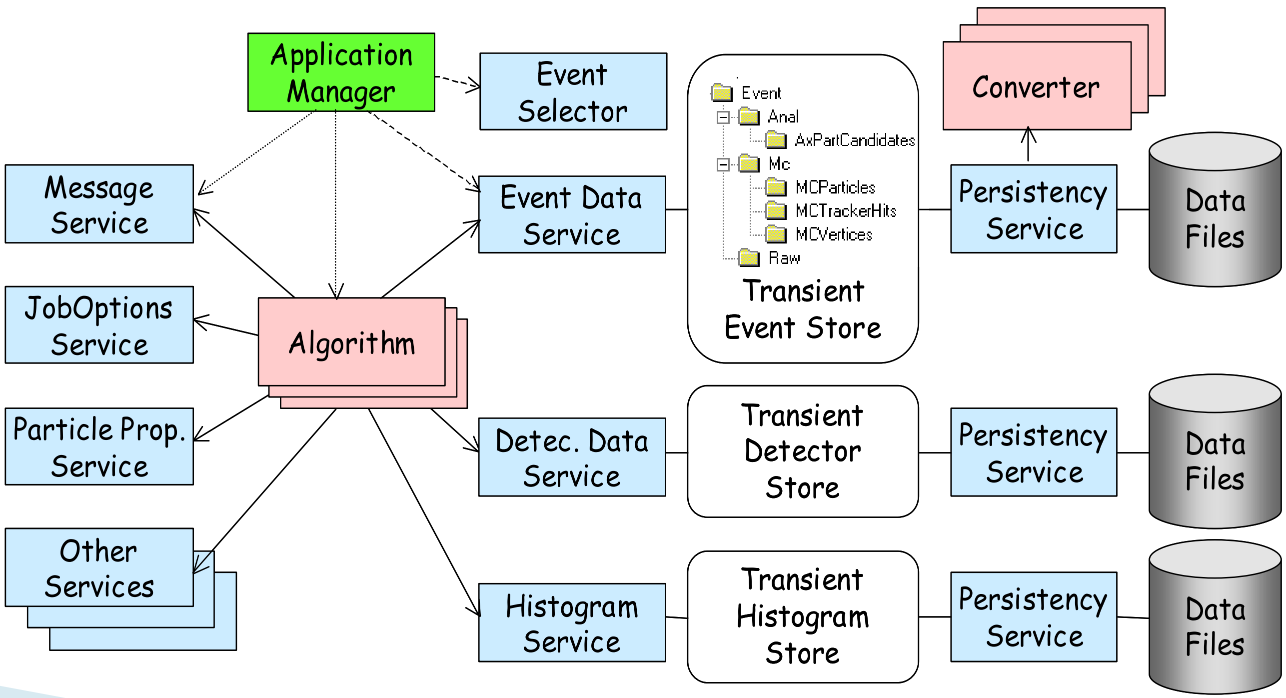
Different style in different domains

- The applications in the different domains may have different emphasis in:
  - Interactivity
  - Database
  - Computation
- Elements of all three are present in all applications



Computation-centric: GAUDI

- Framework adequate for "all" event processing applications
- Algorithms process "event data" with the help of "services" and using "detector data".

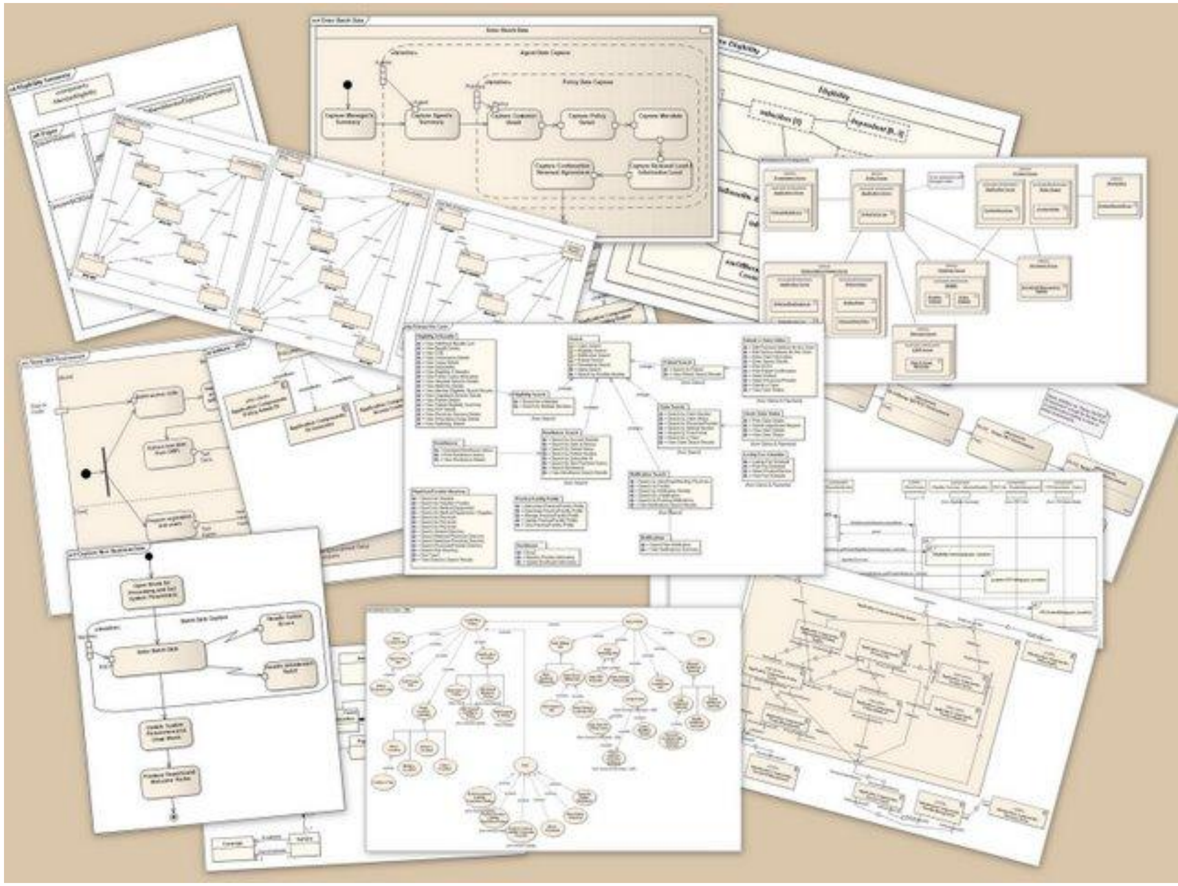


UML

- Unified Modeling Language (UML) is a standardized general-purpose modeling language
- Includes a set of graphical notation techniques to create visual models of software-intensive systems
- Is an open standard
- Supports the entire software development lifecycle
- Supports diverse applications areas
- Is based on experience and needs of the user community
- Supported by many tools

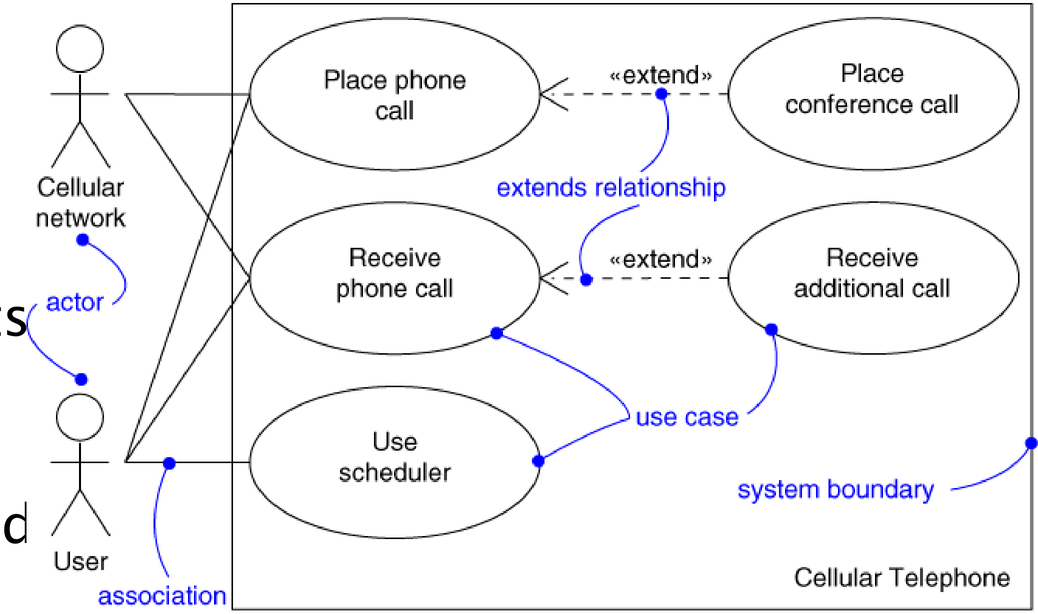
UML Diagrams

- Structure diagrams
  - Class
  - Component
  - Deployment
  - Object
  - Package
- Behavior diagrams
  - Activity
  - State machine
  - Use case
- Interaction diagrams
  - Communication
  - Interaction
  - Sequence



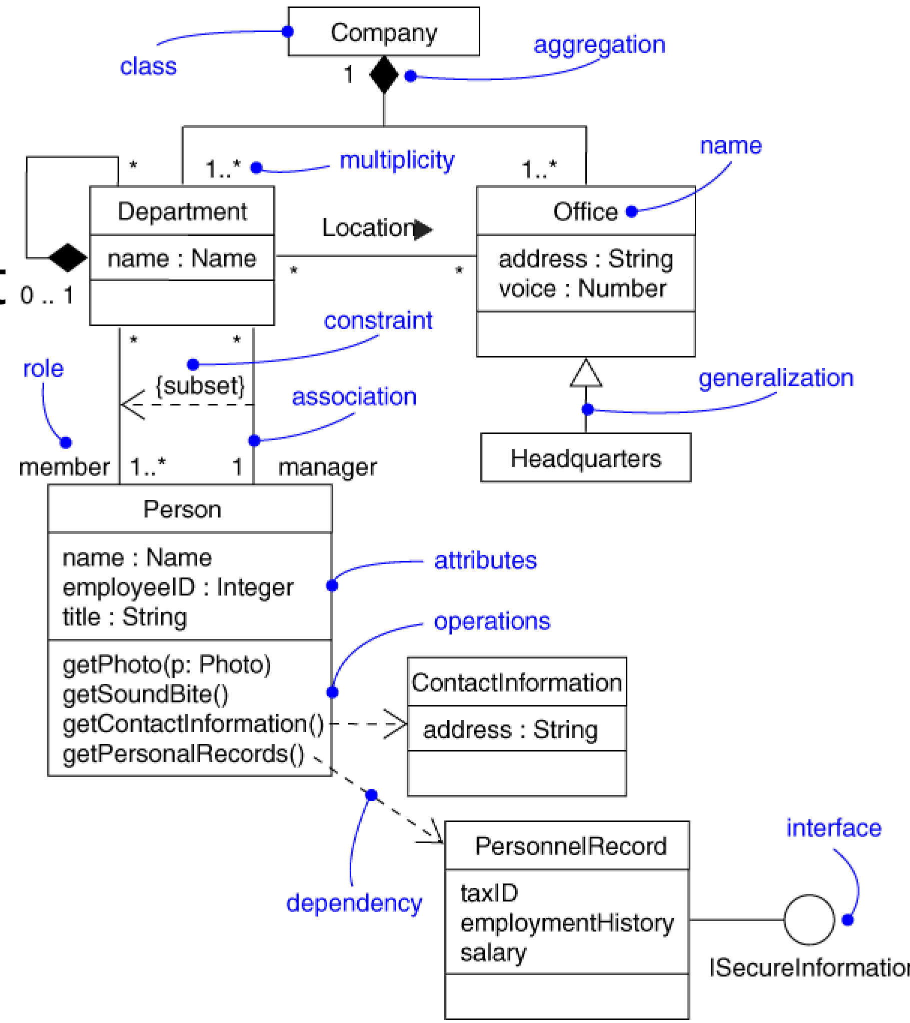
Use Case Diagram

- Captures system functionality as seen by users
- Built in early stages of development
  - Specify the context of a system
  - Capture the requirements
  - Validate a system's architecture
  - Drive implementation and generate test cases
- Developed by analysts and domain experts



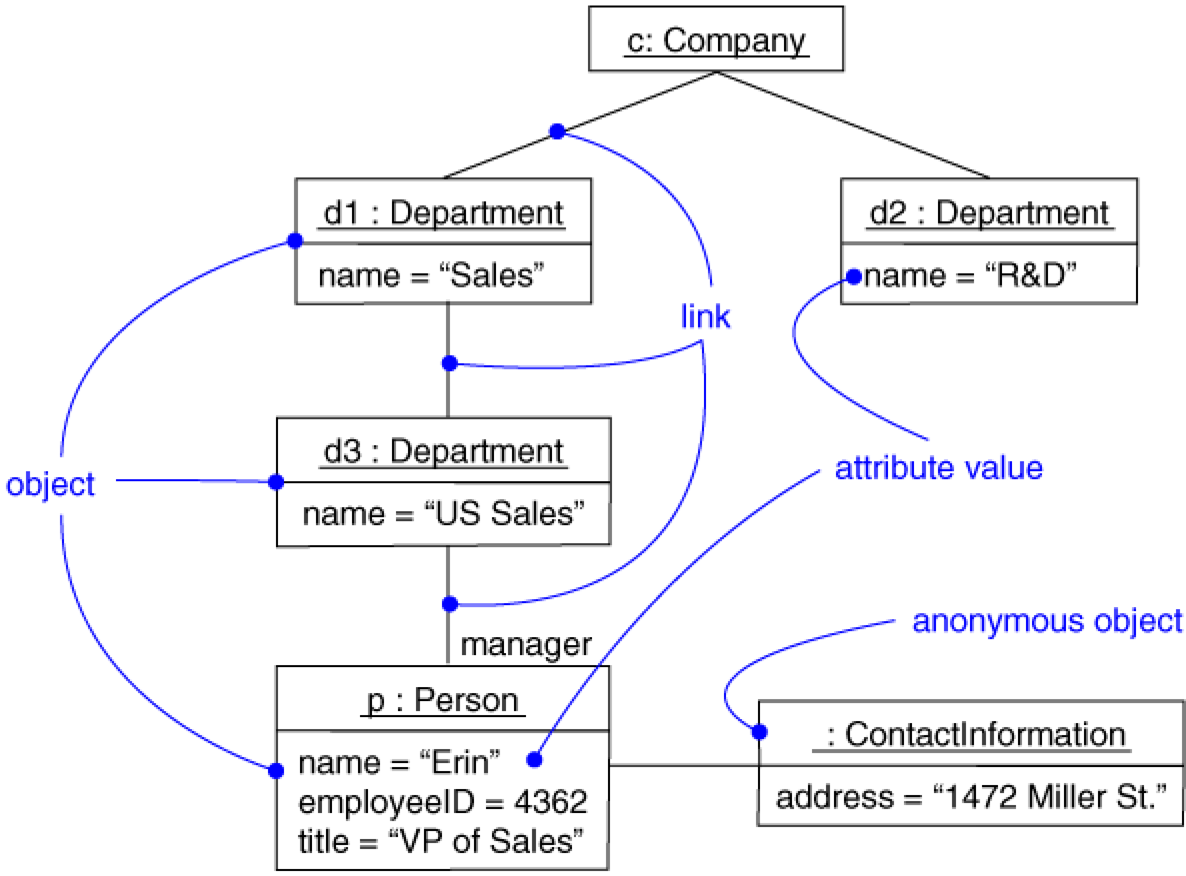
Class Diagram

- Captures the vocabulary of a system
- Built and refined throughout development
  - Name and model concepts in the system
  - Specify collaborations
  - Specify logical database schemas
- Developed by analysts, designers, and implementers



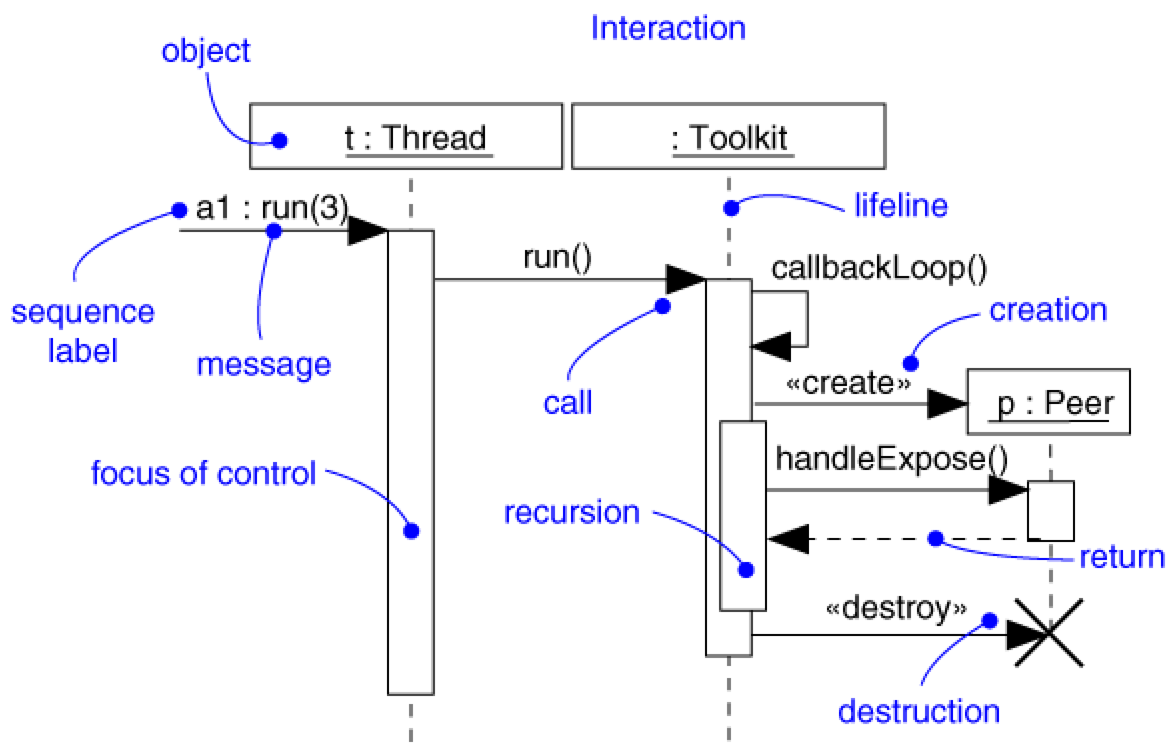
Object Diagram

- Shows instances and links
- Built during analysis and design
  - Illustrate data/ object structures
  - Specify snapshots
- Developed by analysts, designers, and implementers



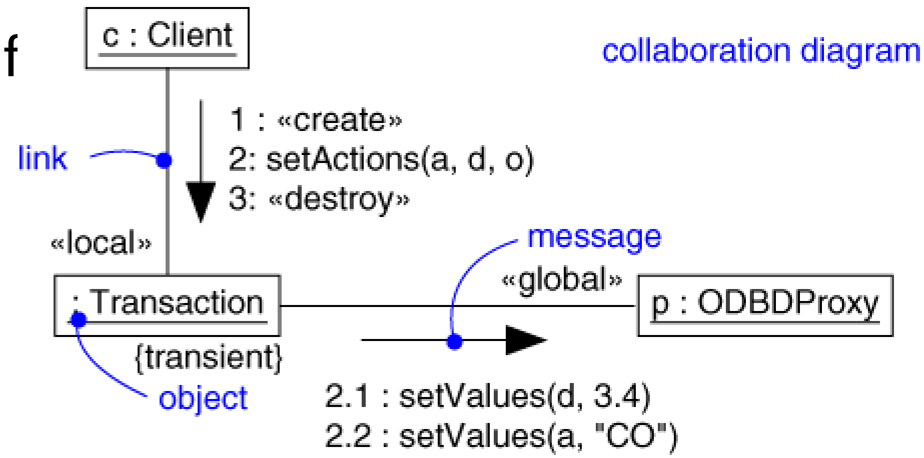
Sequence Diagram

- Captures dynamic behavior (time-oriented)
- Purpose
  - Model flow of control
  - Illustrate typical scenarios



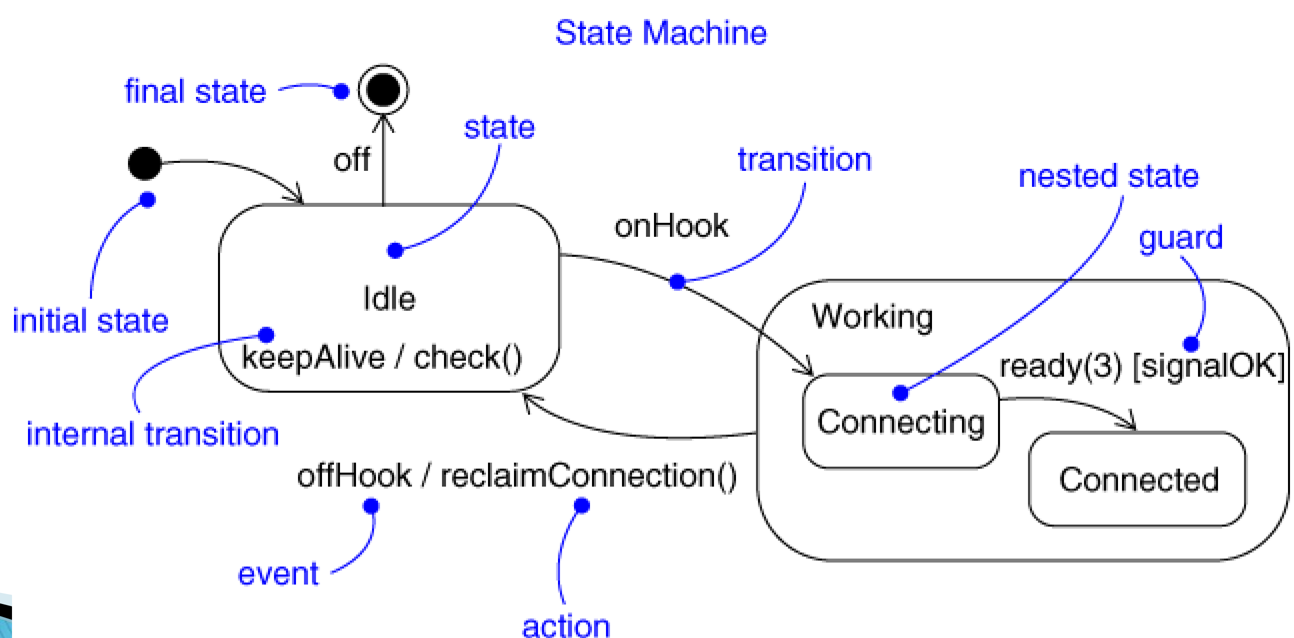
Collaboration Diagram

- Captures dynamic behavior (message-oriented)
  - Model flow of control
  - Illustrate coordination of object structure and control



Statechart Diagram

- Captures dynamic behavior (event-oriented)
- Purpose
  - Model object lifecycle
  - Model reactive objects (user interfaces, devices, etc.)



The Architect

- Experience
  - In software development
  - In the domain
- Pro-active, goal oriented
- Leadership, authority
- Architecture team
  - Balance between technologists, domain experts, users

The Architect

- Not just a top level designer
  - Need to ensure feasibility
- Not the project manager
  - But "joined at the hip"
- Not a technology expert
  - Purpose of the system, "fit",
- Not a lone scientist
  - Communicator

Software architecture team charter

- Defining the architecture of the software
- Maintaining the architectural integrity of the software
- Assessing technical risks related to the software design
- Proposing the order and contents of the successive iterations
- Consulting services
- Assisting marketing for future product definition
- Facilitating communications between project teams

Architecture is making decisions

The life of a software architect is a long (and sometimes painful) succession of suboptimal decisions made partly in the dark.

Frameworks

Software Framework

- A software framework is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality.
- A software framework is similar to software libraries in that they are reusable abstractions of code wrapped in a well-defined API
  - Typically the framework "calls" the user provided adaptations for specific functionality
- Is the realization of a software architecture and facilitates software re-use

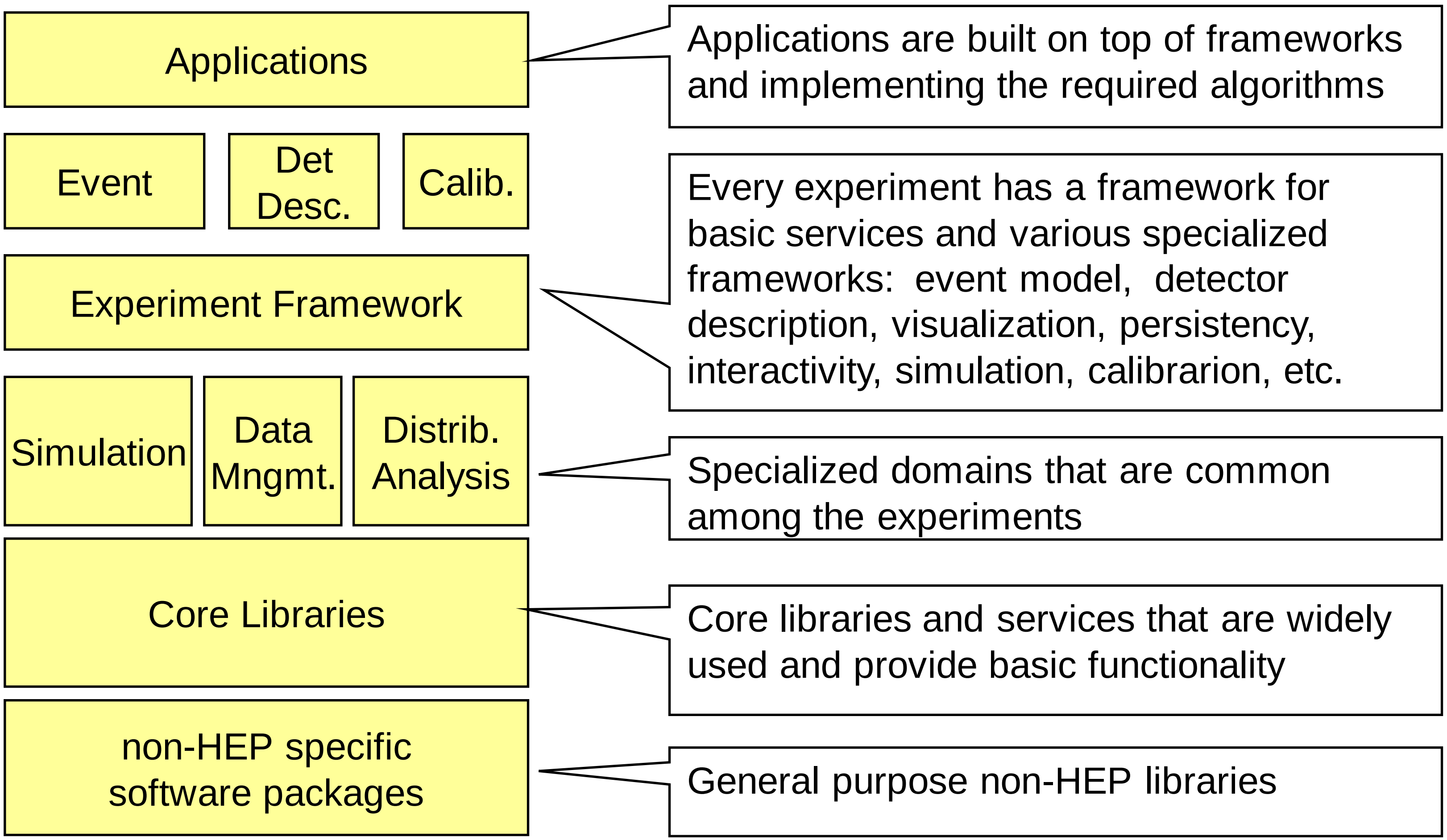
Frameworks in Practice

- A skeleton of an application into which developers plug in their code and provides most of the common functionality

Not a Single Framework

- A single Framework does for fit everywhere
- Each software domain provides its specialized framework
  - E.g. a GUI framework based on signal-slot can be used to build GUI application
- Real complex applications are made typically with a collaboration of frameworks

Software Structure



What is a Framework?

- Framework Definition [1,2]
  - A architectural pattern that codifies a particular domain. It provides the suitable knobs, slots and tabs that permit clients to use and adapt to specific applications within a given range of behavior.
- In practice
  - A skeleton of an application into which developers plug in their code and provides most of the common functionality.

[1] G. Booch, “Object Solutions”, Addison-Wesley 1996

[2] E. Gamma, et al., “Design Patterns”, Addison-Wesley 1995

Framework Benefits

- Common vocabulary, better specifications of what needs to be done, better understanding of the system.
- Low coupling between concurrent developments. Smooth integration. Organization of the development.
- Robustness, resilient to change (change-tolerant).
- Fostering code re-use

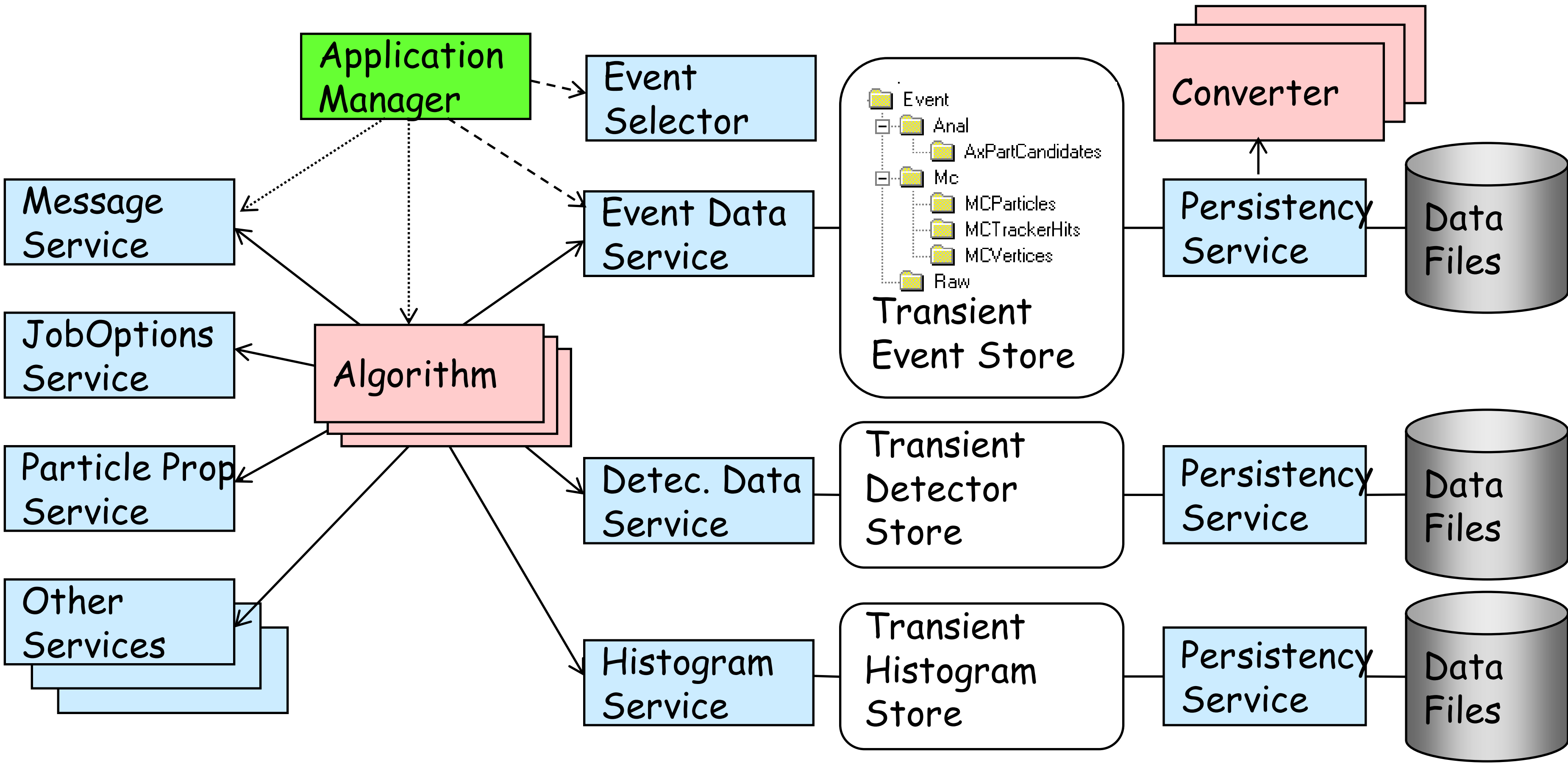
Gaudi Architecture and Framework

An Example of Framework for HEP Applications

Principal Design Choices

- Separation between “data” and “algorithms”
- Three basic categories of “data”
  - event data, detector data, statistical data
- Separation between “transient” and “persistent” representations of data
- Data store-centered (“blackboard”) architectural style
- “User code” encapsulated in few specific places
- Well defined component “interfaces” with plug-in capabilities

Gaudi Object Diagram



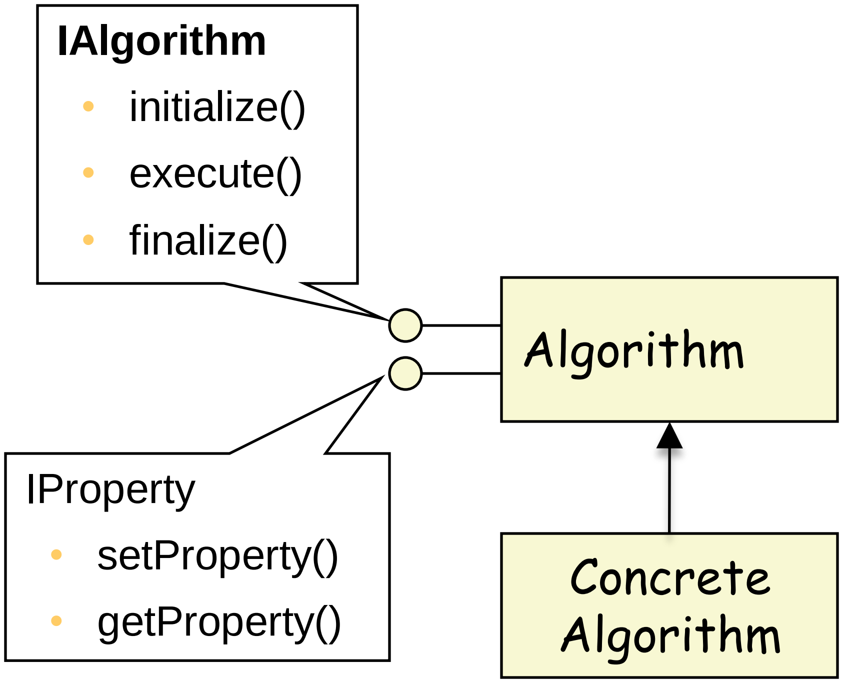
Definition of Terms

- Algorithm
  - Atomic data processing unit (visible & controlled by framework)
- Algorithm Tool
  - Class called by the Algorithm or another Tool to perform a specific function (private and public)
- Data Object
  - Atomic data unit (visible and managed by transient data store)
- Transient Data Store
  - Central service and repository for data objects (data location, life cycle, load on demand, …)

Definition of Terms (2)

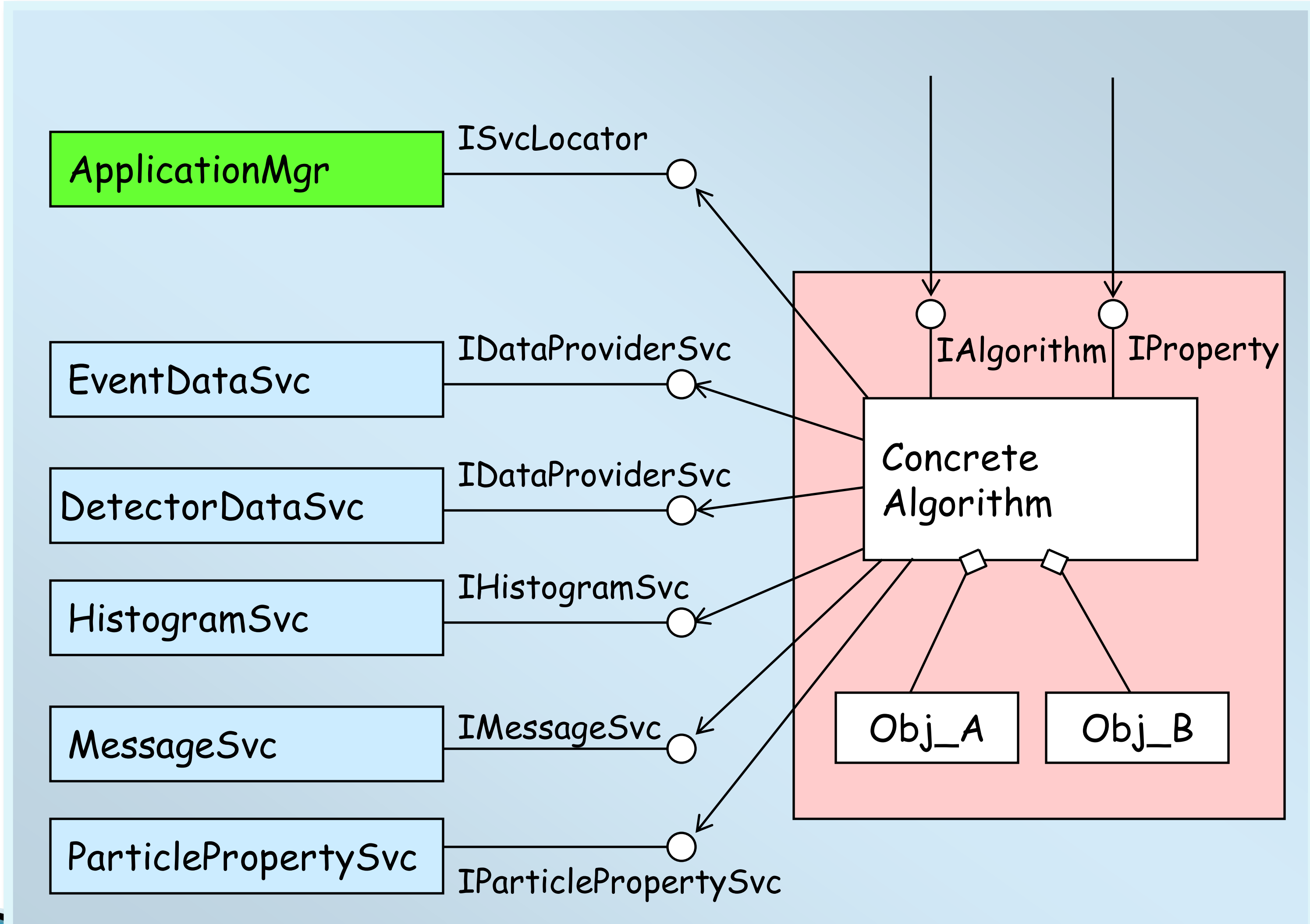
- Services
  - Globally available software components providing framework functionality
- Data Converter
  - Provides explicit/implicit conversion from/to persistent data format to/from transient data
- Properties
  - Control and data parameters for Algorithms and Services

Algorithm



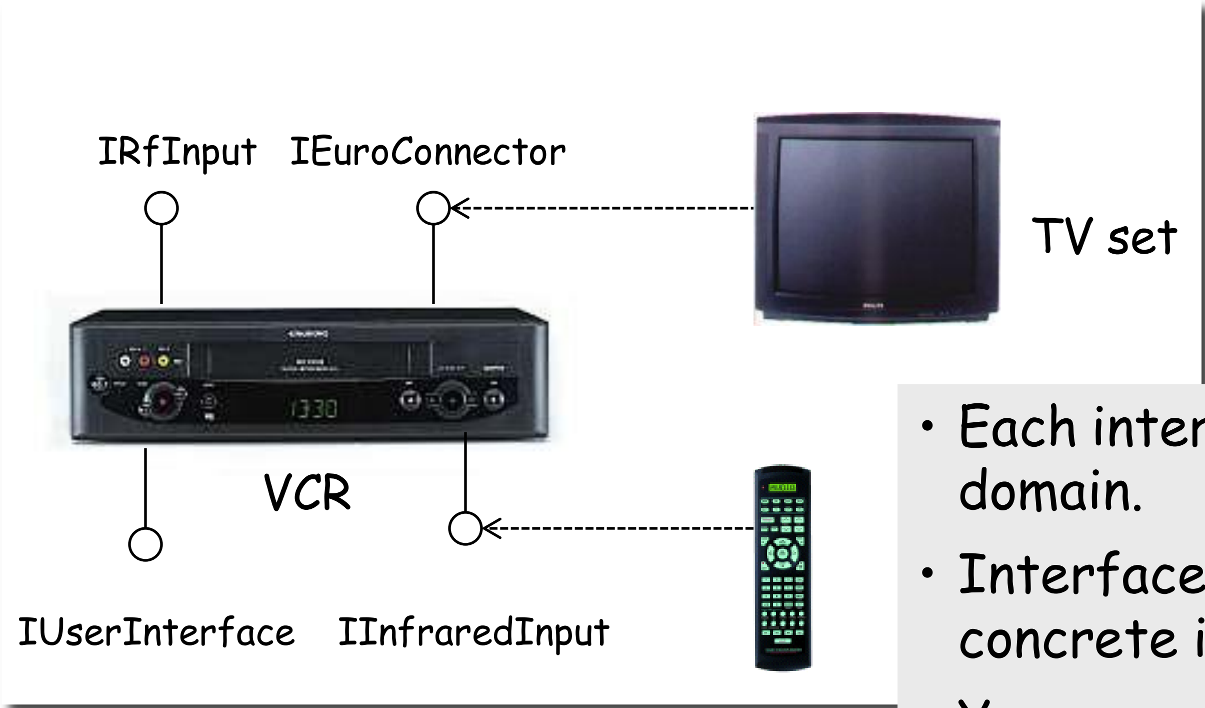
- Users write Concrete Algorithms
- It is called once per physics event
- Implements three methods in addition to the constructor and destructor
  - initialize(), execute(), finalize()

Interfaces



ApplicationMgr
ISvcLocator
EventDataSvc
IDataProviderSvc
DetectorDataSvc
IDataProviderSvc
HistogramSvc
IHistogramSvc
MessageSvc
IMessageSvc
ParticlePropertySvc
IParticlePropertySvc
IAlgorithm
IProperty
Concrete
Algorithm
Obj_A
Obj_B

VCR Interface Model



- Each interface is specialized in a domain.
- Interfaces are independent of concrete implementations.
- You can mix devices from several constructors.
- Application built by composing.
- Standardizing on the interfaces gives us big leverage.

Interfaces in Practice

IMyInterace.h

```
class IMyInterface {
  void doSomething( int a, double b ) = 0;
}
```

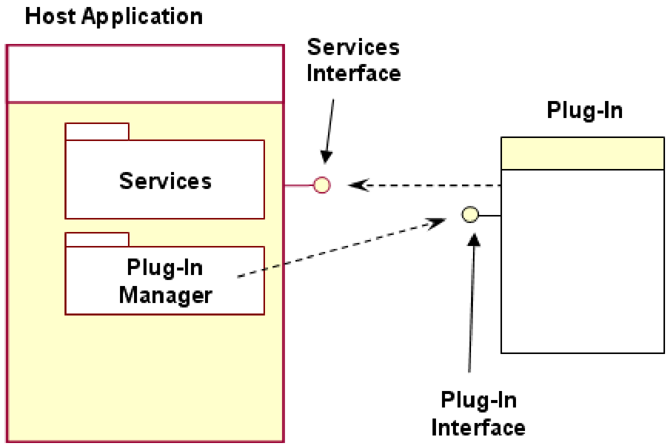
ClientAlgorihtm.cpp

```
#include "IMyInterface.h"

ClientAlgotihm::myMethod() {
  // Declare the interface
  IMyInterface* myinterface;
  // Get the interface from somewhere
  service("MyServiceProvider", myinterface );
  // Use the interface
  myinterface->doSomething( 10, 100.5);
}
```

Plug-ins

- Program extensions to provide a certain, usually very specific function "on demand"
- Applications/frameworks support plug-ins for many reasons (in HEP)
  - to enable third-party developers to create capabilities to extend an application
  - to support features yet unforeseen
  - to reduce the size of the basic application



Reflex Plug-in Service

- Coding the plugin/component
  - No predefined model
  - Declaring factory with signature
- Creating the rootmap file
  - Text file listing all plugins and the associated dynamic library
  - Created with the genmap tool
- Instantiating the plugin
  - Library loaded if needed
  - Strong argument type checking
  - No implementation dependency

MyClass.h

```
class MyClass : public ICommon {
  MyClass(int, ISvc*);
  ...
};
```

MyClass.cpp

```
PLUGINSVC_FACTORY(MyClass,ICommon*(int,ISvc*));
/* implementation */
```

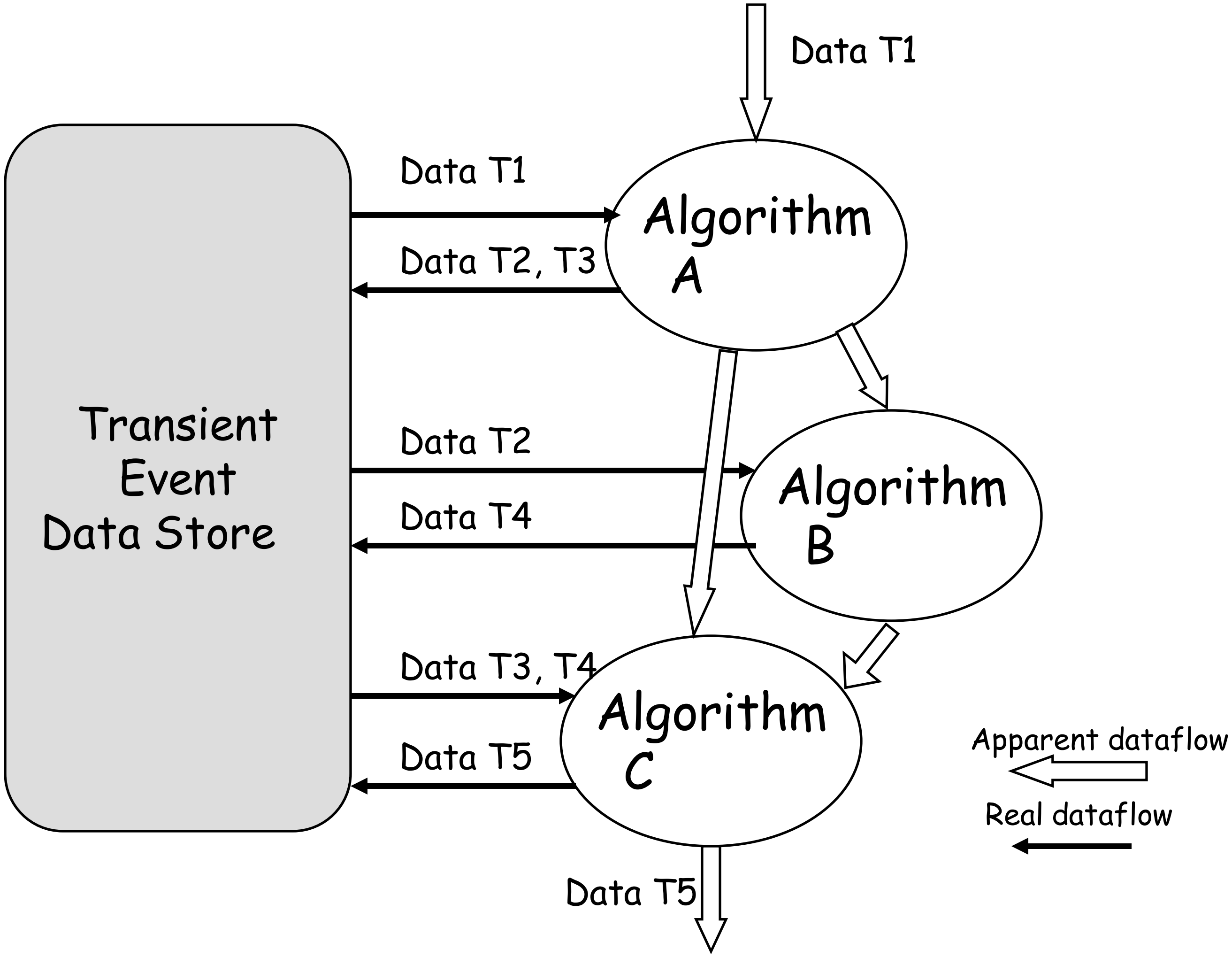
rootmap

```
Library.MyClass:          MyLibrary.so
Library.AnotherClass:     MyLibrary.so
```

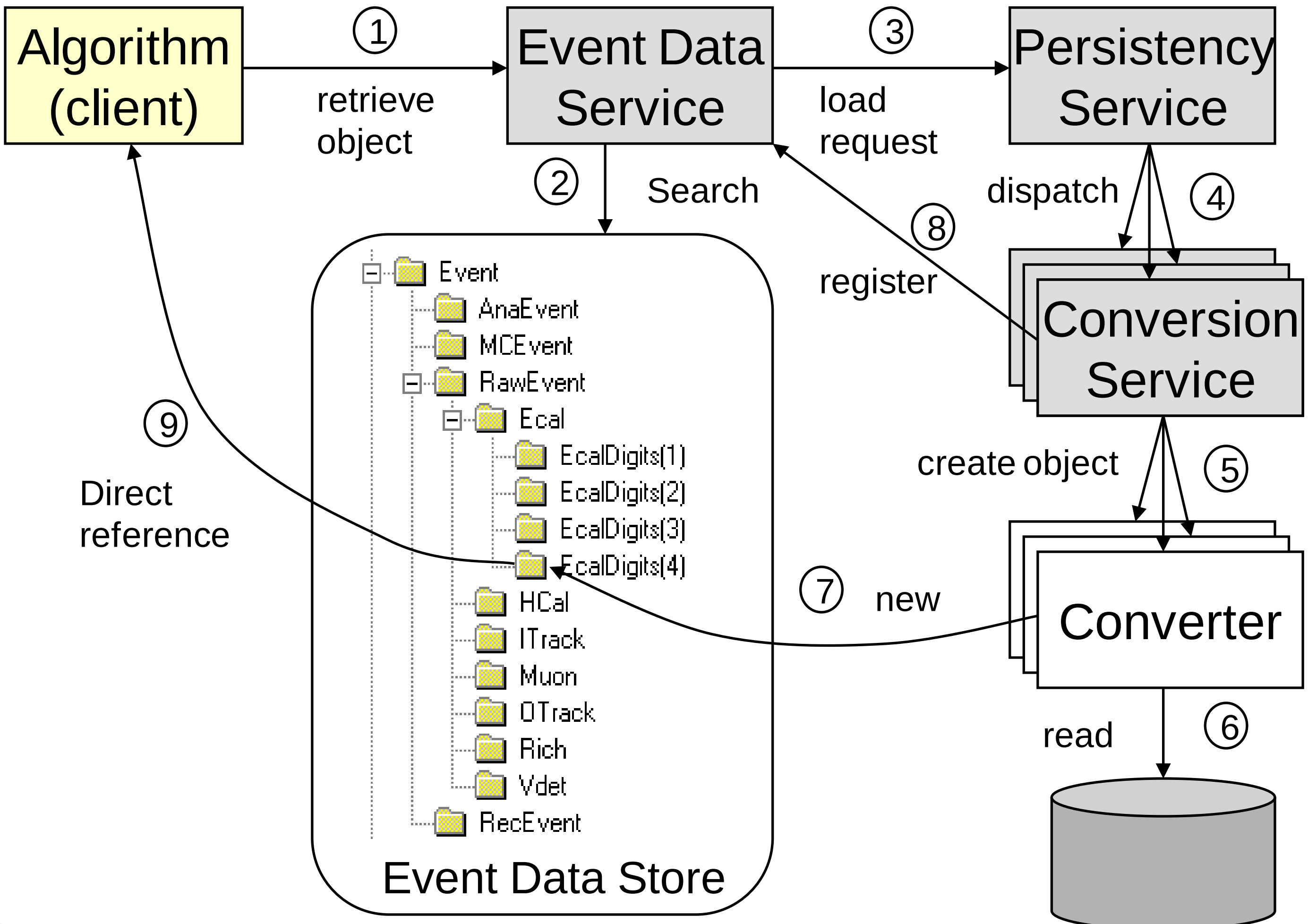
Program.cpp

```
...
ISvc* svc = ...
ICommon* myc;
myc = PluginSvc::create<ICommon*>("MyClass",10, svc);
if ( myc ) {
  myc->doSomething();
}
```

Algorithm & Transient Store

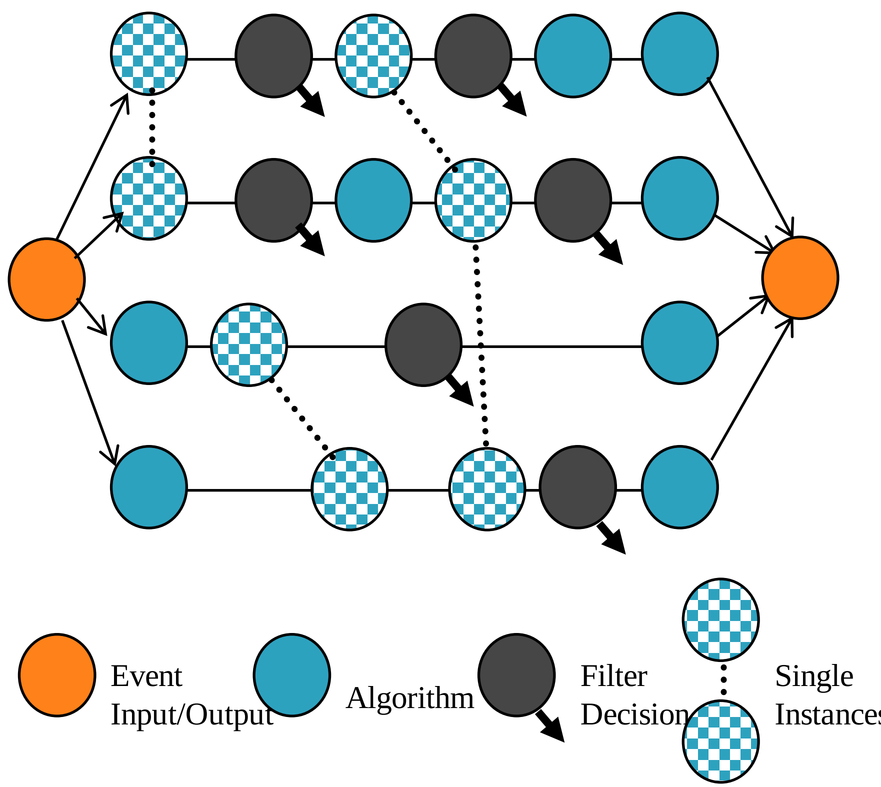


Loading Transient Store



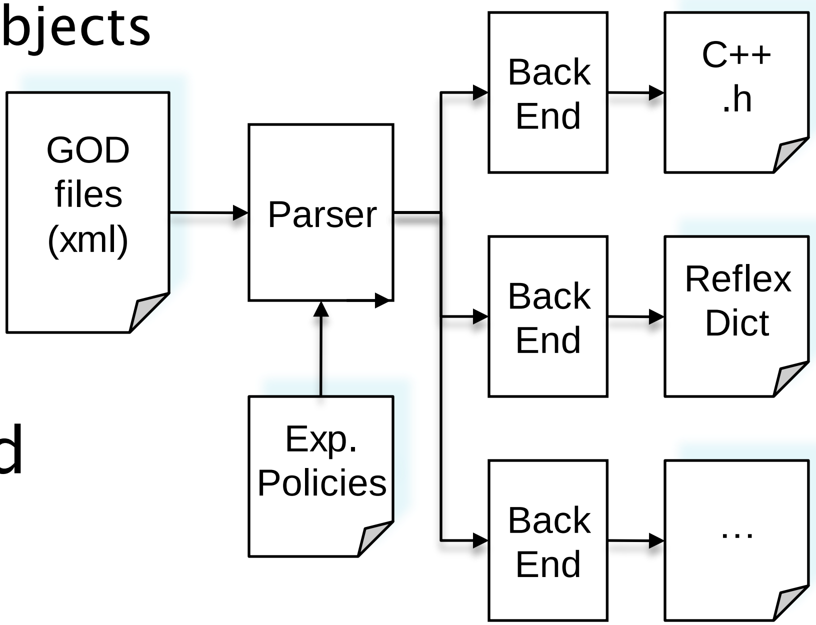
Complex Control Sequences

- Concept of sequences of *Algorithms* to allow processing based on physics signature
  - Avoid re-calling same algorithm on same event
  - Different instances of the same algorithm possible
- Event filtering
  - Avoid passing all the events through all the processing chain



Data Object Description

- Definition of objects on a higher level
  - Easy language for defining objects
  - Ability to derive several implementations from this source
  - **Uniform layout of objects**
  - Easily extensible
- Produce C++ headers and Reflex dictionaries automatically
  - **Global optimization possible (e.g. memory pools)**

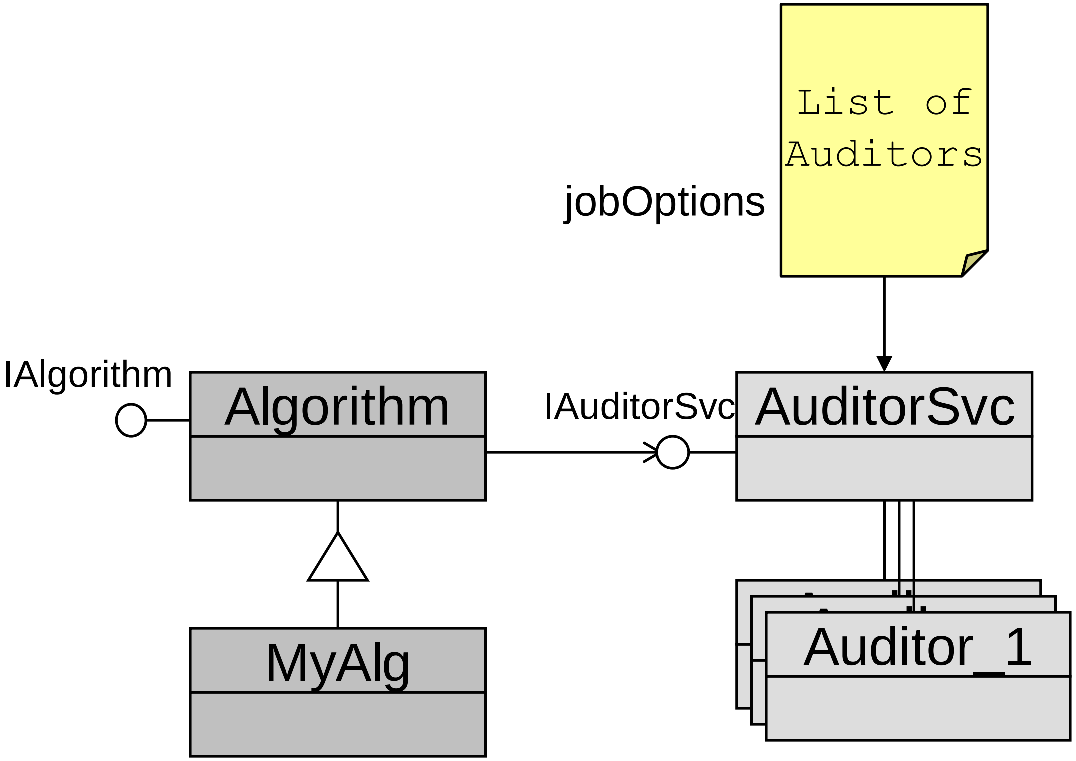


Auditors

- The Auditor Service provides a set of *auditors* that can be used to provide monitoring of various characteristics of the execution of Algorithms
  - *ChronoAuditor*, *MemoryAuditor*, etc.
- Each auditor is called immediately before and after each call to each Algorithm instance
  - Tracks some resource usage of the Algorithm

➔Built-in performance monitoring is essential !!

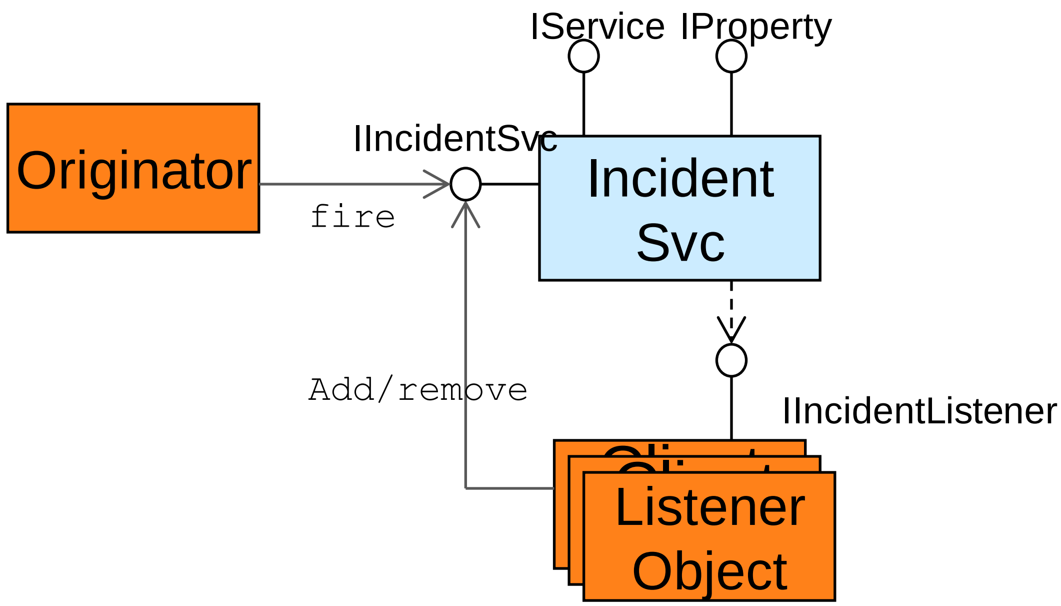
Auditors


List of Auditors
jobOptions
IAlgorithm
Algorithm
IAuditorSvc
AuditorSvc
MyAlg
Auditor_1

Incidents

- The Incident Service provides synchronization facilities to components in a Gaudi application
- *Incidents* are named software events that are generated by software components and that are delivered to other components that have requested to be informed when that incident happens
  - A number of predefined *incidents* such as 'beginRun', 'endEvent', 'openFile'

Incident Service



Data On Demand

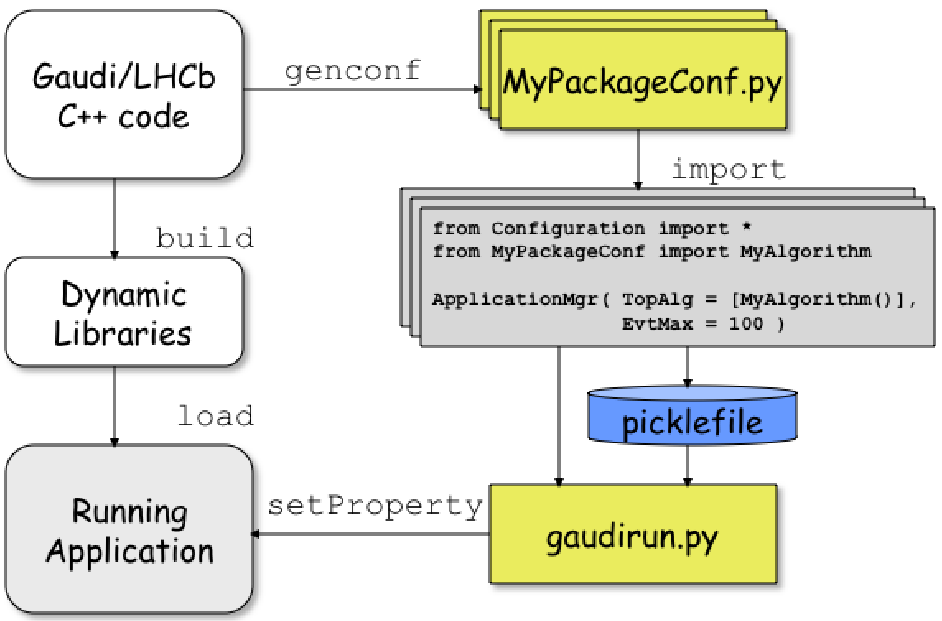
- Typically the execution of Algorithms are explicitly specified by the initial sequence and and sub-sequences
  - Avoid too-late loading of components (HTL)
  - Easier to debug
- For some use-cases it is necessary to trigger the execution of a given Algorithm by accessing an Object in the Transient Store
  - The DataOnDemand Service is can be configured to provide this functionality

Other Gaudi Services

- JobOptions Service
- Message Service
- Particle Properties Service
- Event Data Service
- Histogram Service
- N-tuple Service
- Detector Data Service
- Magnetic Field Service
- Tracking Material Service
- Random Number Generator
- Chrono Service
- (Persistency Services)
- (User Interface & Visualization Services)
- (Geant4 Services)

Configuring the Application

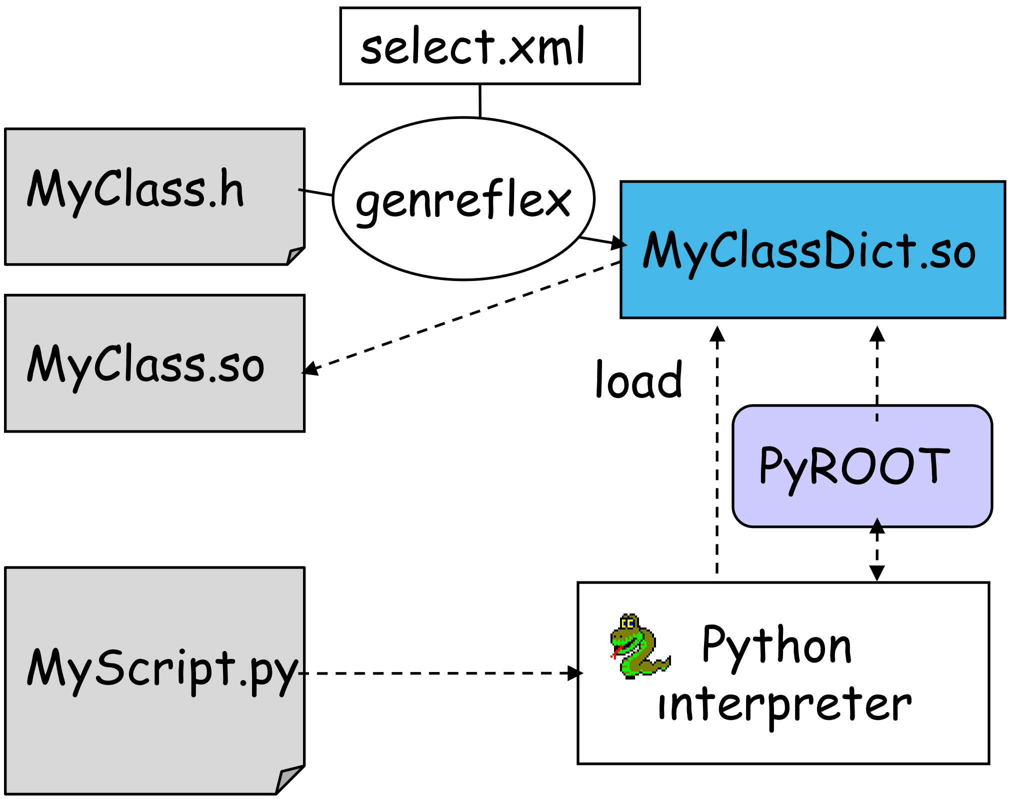
- Each Framework component can be configured by a set of 'properties' (name/ value pairs)
- In total thousands of parameters need to be specified to fully configure a complex HEP application
- Using Python to facilitate the task
  - Build-in type checking



Interactivity and scripting

- Interactivity and scripting are essential use cases for any HEP framework
  - Scripts for rapid prototyping and trying new ideas
  - Testing frameworks
  - GUI applications
- A convenient way to achieve it is to provide bindings to a scripting language such as Python (or a C++ interpreter)
  - Once this is done the rest comes automatically

PyROOT: Mode d'emploi



- From class definitions (.h files) a "dictionary" library is produced
  - Description of the class
  - "stub" functions to class methods
- Absolutely non-intrusive
- The PyROOT module does the adaptation between Python objects and C++ objects in a generic way
  - It works for any dictionary

Summary: Frameworks

- All experiments have developed Software Frameworks
  - General architecture of any event processing applications (simulation, trigger, reconstruction, analysis, etc.)
  - To achieve coherency and to facilitate software re-use
  - Hide technical details to the end-user Physicists
  - Help the Physicists to focus on their physics algorithms
- Applications are developed by customizing the Framework
  - By the “composition” of elemental Algorithms to form complete applications
  - Using third-party components wherever possible and configuring them
- ALICE: AliROOT; ATLAS+LHCb: Athena/Gaudi; CMS: CMSSW

Integrating Technologies

Software Re-use

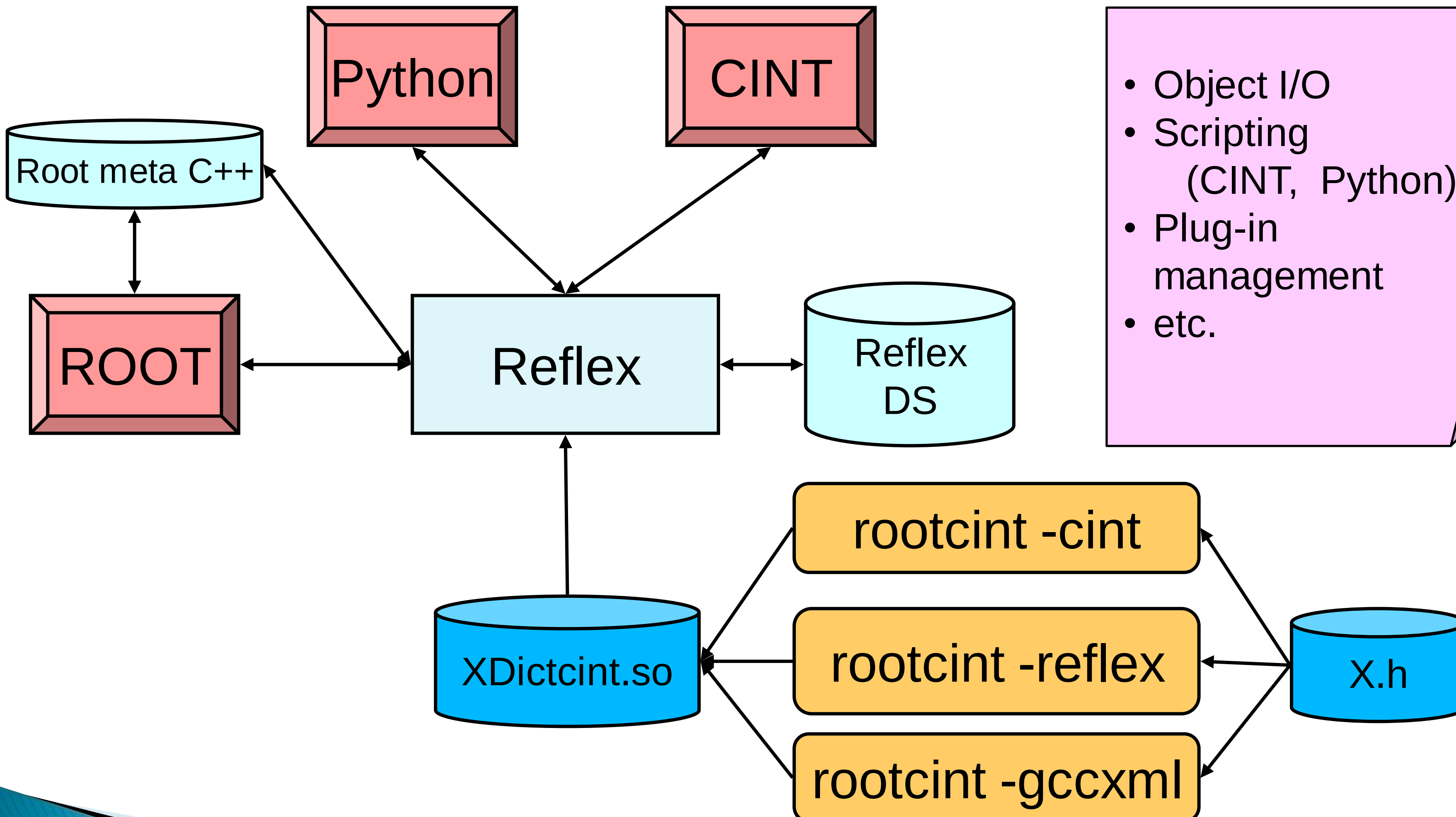
When Frameworks are not Possible

- At occasions you need to a build software system/application made of independently developed components
  - Using existing class libraries
  - They cannot be re-done using a single ‘framework’
  - Building adaptation layers are not always possible and effective
- Examples
  - Integrating MC generators in ROOT
  - Performing ROOT I/O on Geant4 Applications

Software Integration Elements

- Dictionaries
  - Dictionaries provide meta data information (reflection) to allow introspection and interaction of objects in a generic manner
- Scripting languages
  - Interpreted languages are ideal for rapid prototyping
  - They allow integration of independently developed software modules (software bus)
  - Standardizing on CINT and Python scripting languages
- Component model and plugin management
  - Modeling the application as components with well defined interfaces
  - Loading the required functionality at runtime

Strategic role of C++ reflexion

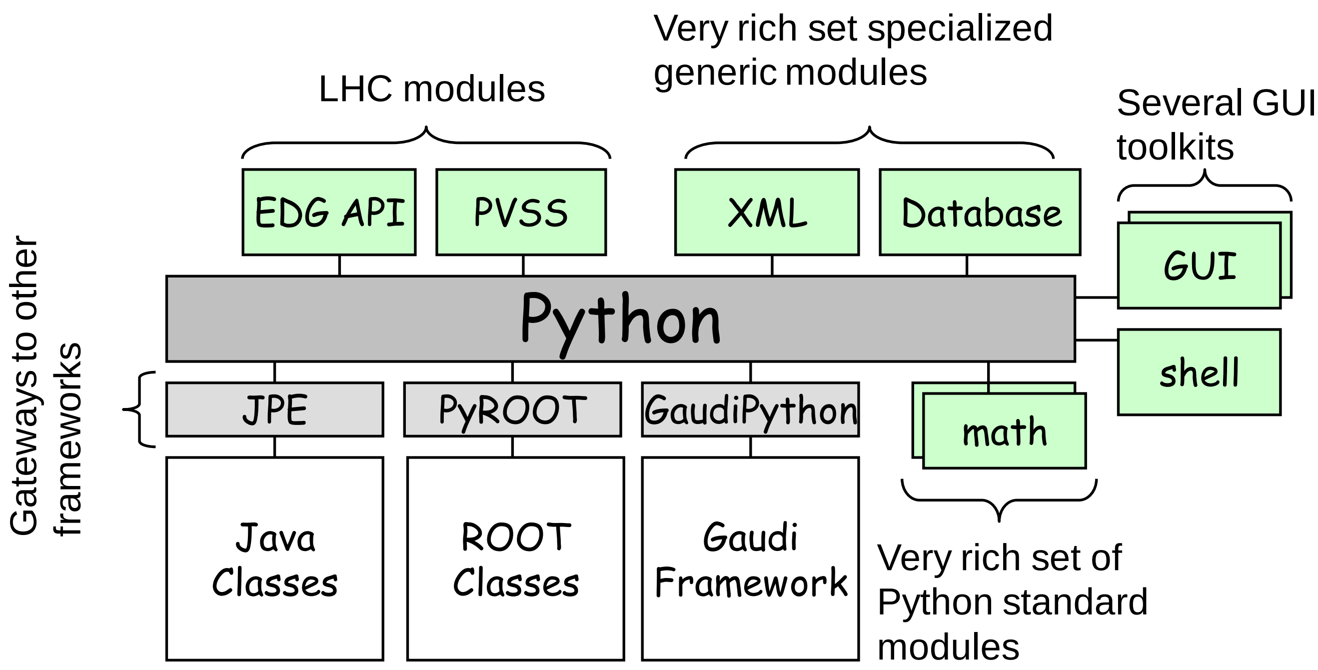


- Object I/O
- Scripting (CINT, Python)
- Plug-in management
- etc.

Python <-> C++ Interoperation

- The bulk of code for the new HEP experiments is written in C++
  - Still some portions of FORTRAN with plans to migrate
  - Java and other languages almost non-existent
- Need Python *bindings* to C++ code
  - Hand-written (C-API) or generated
  - Requires taking care of:
    - Object, parameter conversions
    - Memory management
    - C++ function overloading
    - C++ templates
    - Inheritance and function callbacks

Python as Software "Bus"



Summary

- Introduced the main concepts of software architecture
  - Why it is needed, what it means, modeling concepts and languages (UML), etc.
  - The role of architect
- Introduced software frameworks and their hierarchy
- Used GAUDI framework as an example of HEP event data processing framework
  - The main design criteria
  - Introduction to few of the main concepts and functionalities
- Software integration elements

References

- **Grady Booch, *Object Solutions*, Addison-Wesley, 1995.**
- Eric Gamma, John Vlissides, Richard Helm, Ralph Johnson, Design Patterns, Addison-Wesley 1995.
- ***Rational Unified Process* 5.0, Rational, Cupertino, CA, 1998**
- Len Bass, Paul Clements & Rick Kazman, Software Architecture in Practice, Addison-Wesley, 1998