# Control States
# for the Atlas Software Framework

Paolo Calafiura, LBL

ACAT 2000

Fermi Lab, October 19 2000

# The Atlas Software Architecture

â Overall design principles specified Dec 99 by the Architecture Task Force

– data and algorithm object separation

– proxy data access using a “Transient Data Store”

– no direct module-to-module communication

– traditional control flow

– technology-independent database access layer

â Athena Framework prototype implementation

– based on the existing Gaudi architecture effort (initiated by LHCb)

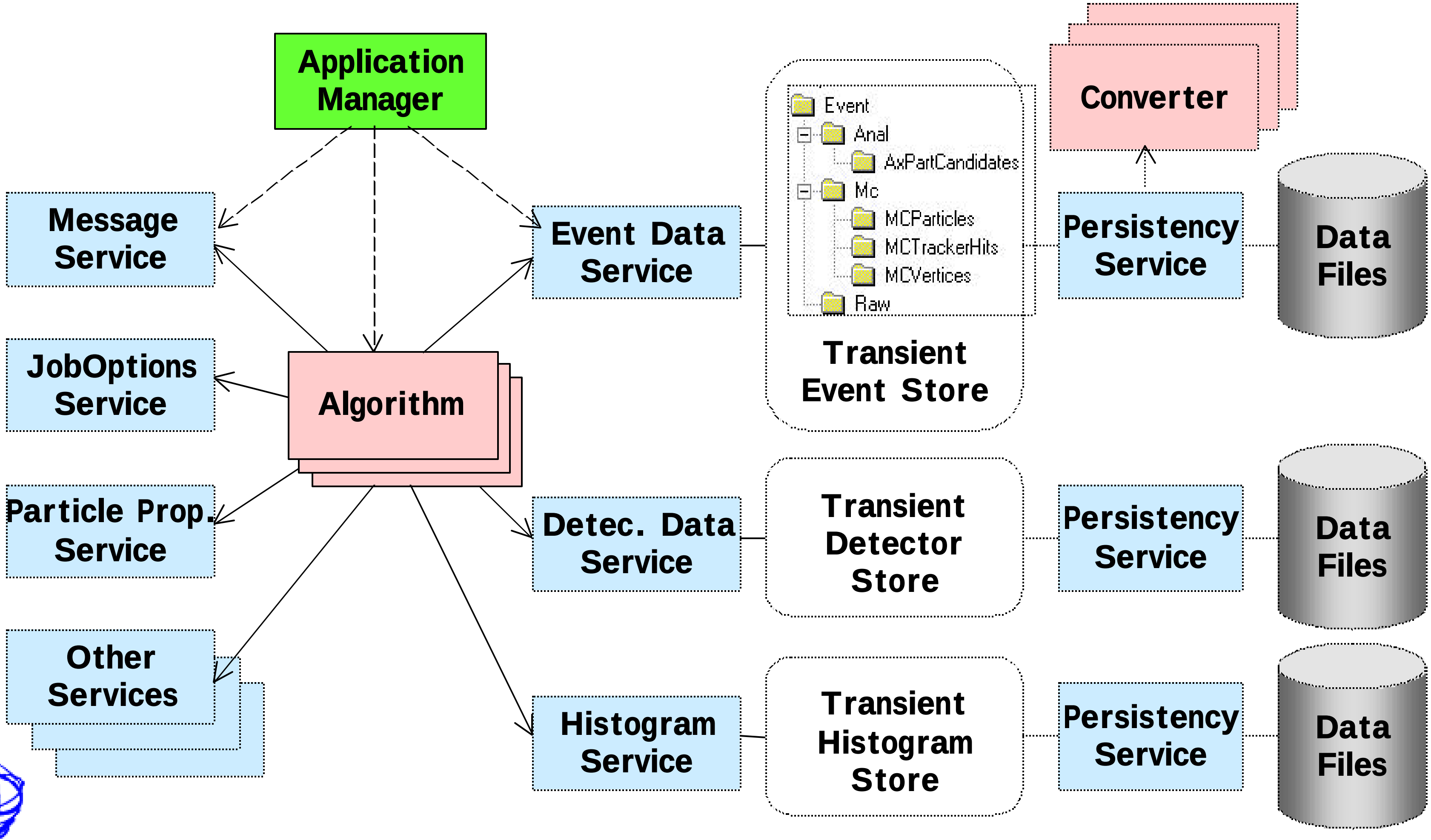
# Control Framework

The control framework is the part of an infrastructure that makes sure that

- The right piece of software
- Runs
- At the right time
- With the right inputs and
- The outputs go to the right place

(Lassi Tuura)

# GAUDI Architecture



# Definitions

â Algorithm (Module)

– Atomic unit (visible & controlled by framework) of calculation and/or processing.

â ApplicationMgr

– creates and initializes Services and Algos. Drives the Algorithms processing

â Data Object (Collection)

– Atomic unit (visible & managed by transient data store) of data. NOT necessarily a dumb data object.

â Transient Event (Data) Store

– Central service and repository for data objects. Provides data location, data object life cycle management, transparent smart pointer/data converter interaction.

– Also Transient Histogram & Detector Stores

â Data Converter

– Provides explicit (some implicit soon) conversion from “arbitrary” persistent data format (ie. ZEBRA, Objectivity, etc.) to transient data object.

â Services

– Globally available software components providing framework functionality.

â Properties

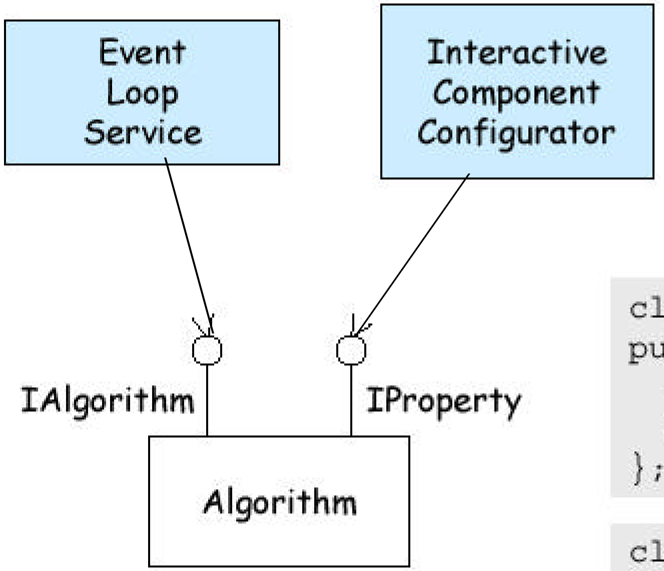
– Control and data parameters for Algorithms and Services.

â Job Options File

– Text file defining configuration and properties.

**(from Craig Tull’s Gaudi Tutorial Introduction)**

# Interface Model (2)



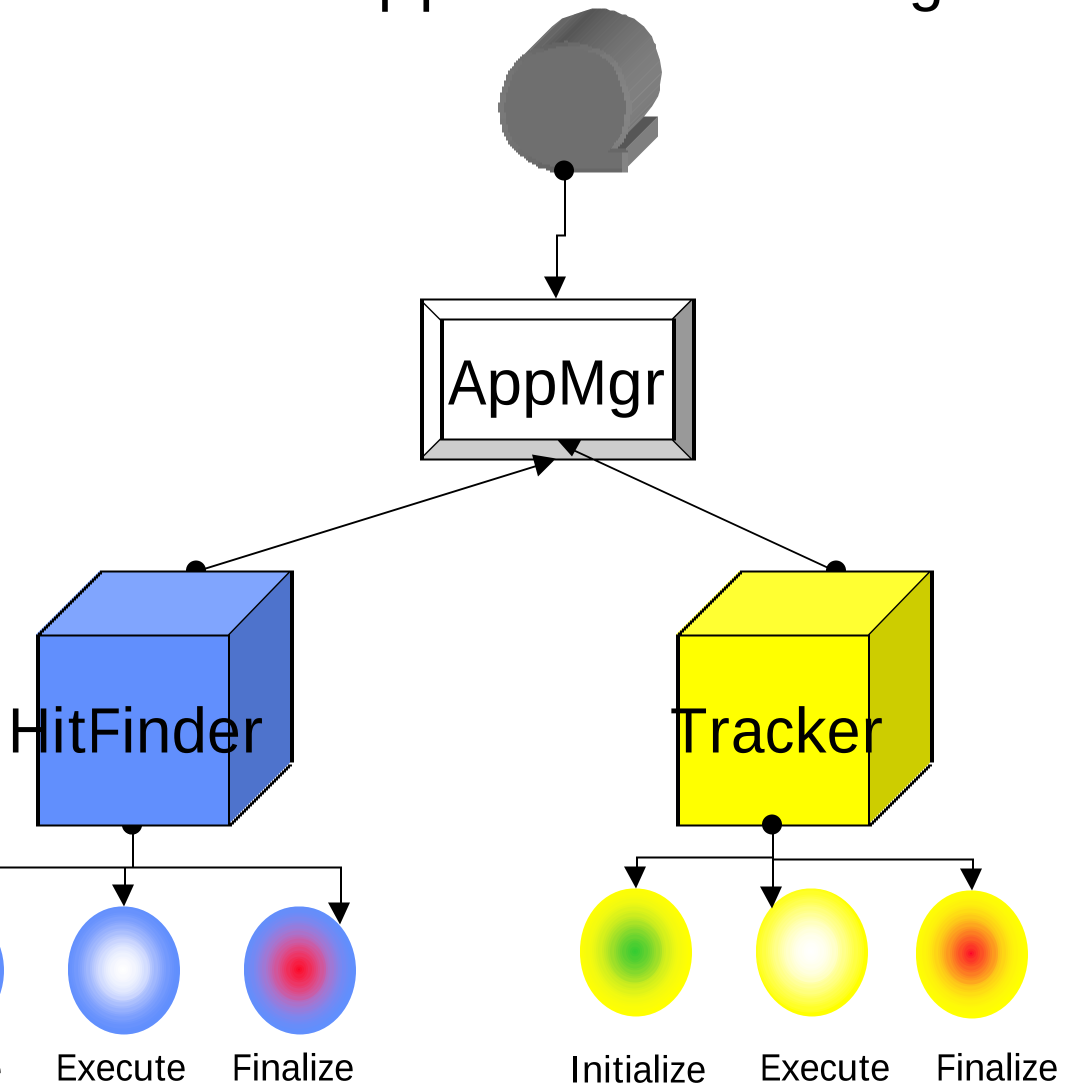
```
class IAlgorithm : virtual public IInterface {
 public:
  virtual StatusCode initialize() = 0;
  virtual StatusCode execute() = 0;
  virtual StatusCode finalize() = 0;

  virtual const std::string& name() const = 0;
  virtual StatusCode sysInitialize() = 0;
  virtual StatusCode sysFinalize() = 0;
};
```

```
class IProperty : virtual public IInterface  {
public:
  virtual StatusCode setProperty(const Property& p) = 0;
  virtual StatusCode getProperty(Property *p) const = 0;
};
```

```
class Algorithm : virtual public IAlgorithm,
                  virtual public IProperty {
public:
 ...
}
```

# The Application Manager



- Input
- Algorithms
- State Methods

# What’s missing?

â For most use cases, nothing really...

â Use cases not easily covered by this approach

- – Event filtering: I/O modules must handle disk file open/close actions
- – Calibration: must handle stepping of input signal
- – Simulation: pile-up of events coming from multiple streams

â Don’t want to require each algorithm to handle a “file opened” action

â Too much coupling among Algos and the ApplicationMgr:

- – each and every algo must implement exacty three transitions: initialize, execute and finalize

# What else do we need?

â Separate control from access to core services

â Support multiple event sources (e.g. for pile-up studies)

â Notify modules only about the transitions they may be interested into

– Notification must be type-safe: only modules implementing the “right” state transition interface can be notified.

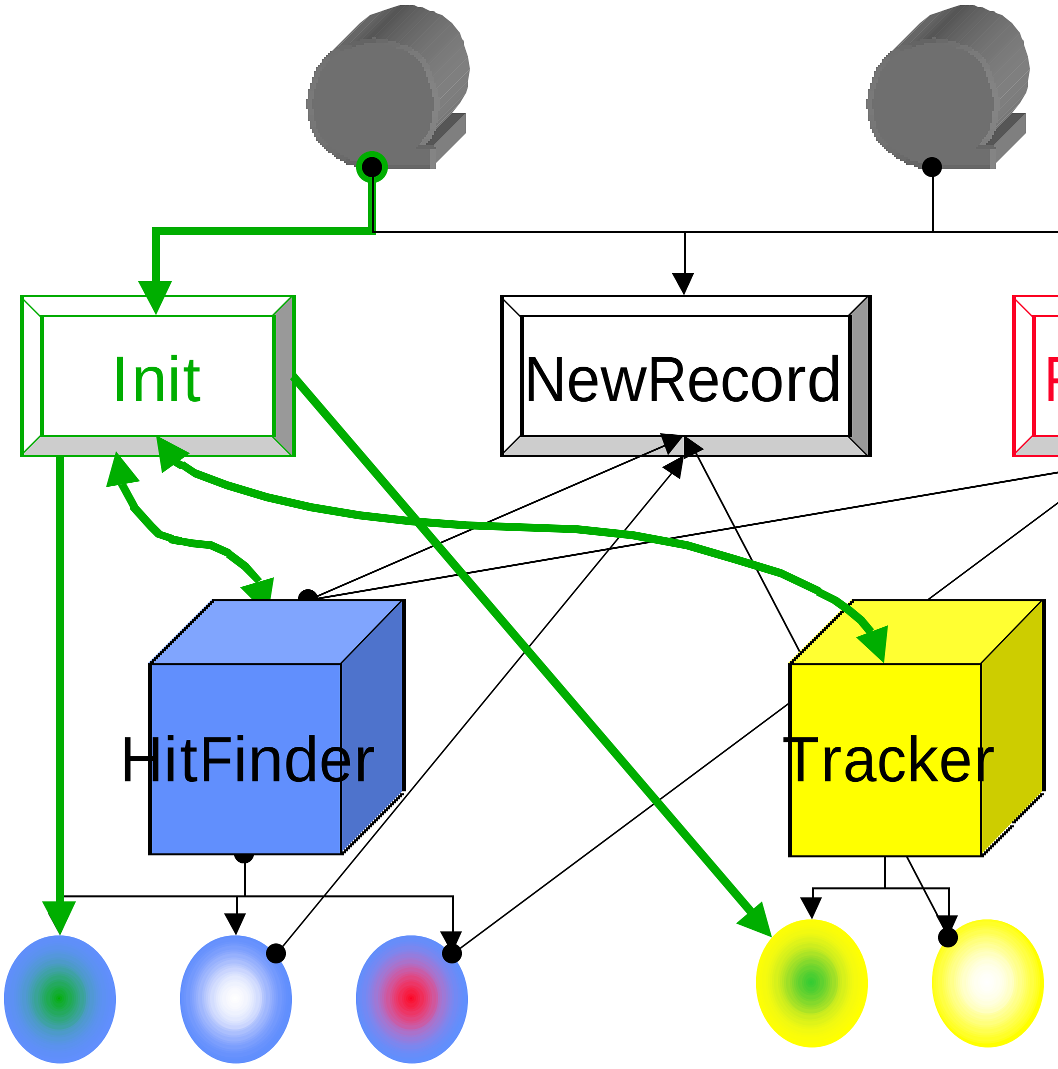
â Control the order in which the modules handle actions

â Define the states, the order of modules and the state sources, dynamically via the User Interface

â No physical coupling between ApplicationMgr and modules:

– states can be added or removed without triggering massive recompilations

# The Control States Framework



Init

NewRecord

Finalize

HitFinder

Tracker

- Sources
- States
- Modules
- State Methods

# The Core Classes

â State Source

- drive the framework generating actions

â State (and Concrete States)

- observe sources for matching actions, run module methods

â Modules

- handle state transitions,
  adding matching state methods to their queues

# Implementation

â As usual we added a level of indirection (actually two):

- each source is an Observable generating actions (=states)
- each state is a typed Observable that notifies its registered modules when it observes the corresponding action
  (in a sense each state is a separate control framework)

â A Module implements a separate interface for each action he can handle

â It looks very much like the Typed Message/MultiCast pattern (J. Vlissides, “Pattern Hatching”, great book)

# Scenario: Running a State

- The source notifies all registered states that he has a newRecord action

  ```
  StateSource::notify DEBUG: notifying newRecord
  ```

- newRecord state catches the action and notifies its observers, the managers

  ```
  State::update DEBUG: newRecord[instanceof NewRecordState] got
  message newRecord
  ```

- Each manager add the matching method to the state queue
- Now newRecord runs the scheduled methods

  ```
  State::run DEBUG: newRecord[instanceof NewRecordState] starts
  Hitfinder::newRecord DEBUG: running
  State::run WARNING:
  newRecord[instanceofHitFinder::__newRecord] was not ready and
  had to be rescheduled
  Histogrammer::newRecord DEBUG: running
  Hitfinder::newRecord DEBUG: running
  ```

# Where do we stand?

â We have a web page
http://electra.lbl.gov/ATLAS/framework/controlstates/actiondesign.html

â We have a prototype

- Integrated in Atlas SRT
  - can get a stand alone version from URL above
- Integration with the ApplicationMgr (being rewritten) in progress

â Use it to explore interactions with other new domains

- Scripting/User Interface
- Event Data Model (next talk, VLSC305)

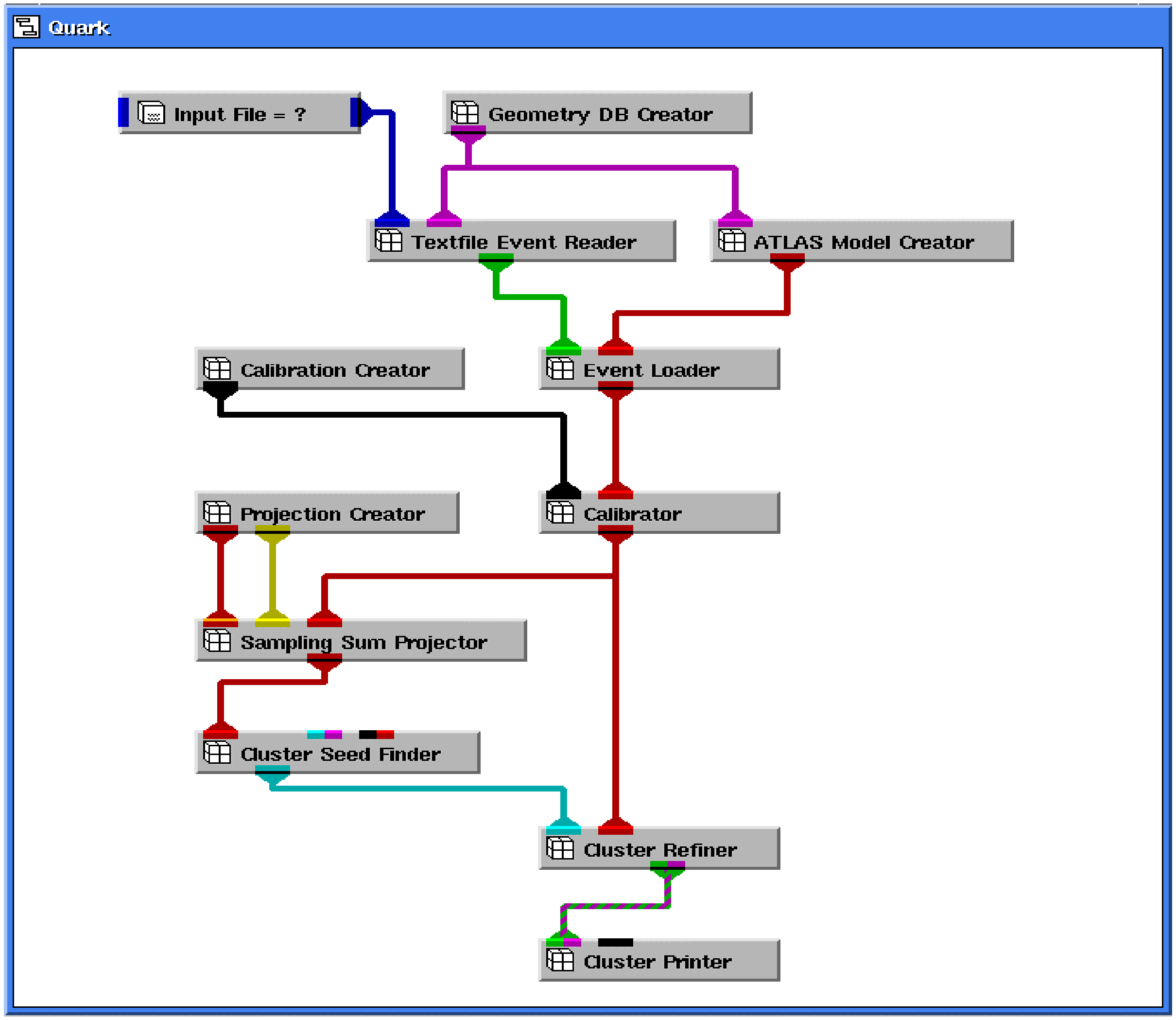
# Thanks to

â So many people that I’ll sure forget some:

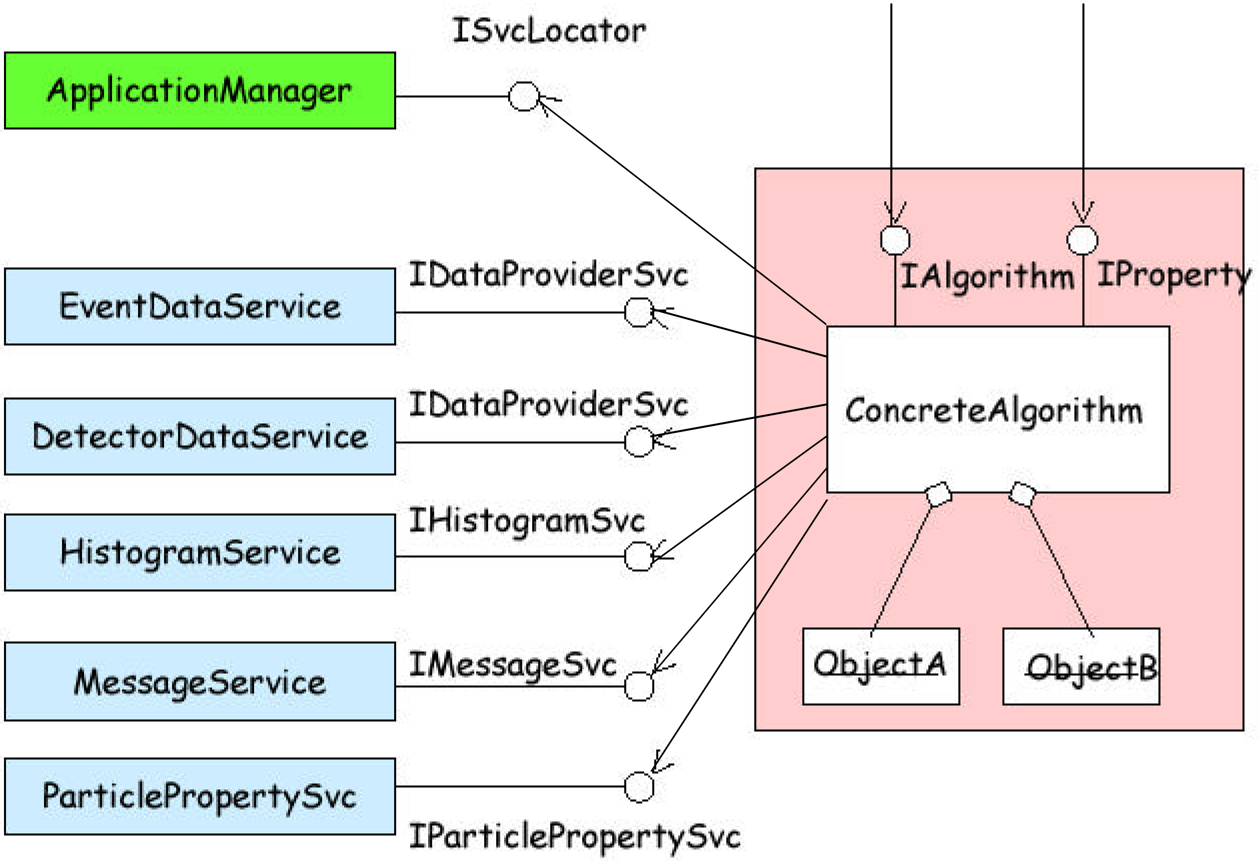
- Vincenzo Innocente
- Jim Kowalkowski
- Charles Leggett
- Pere Mato
- John Milford
- Dave Quarrie
- Marjorie Shapiro
- Lassi Tuura
- Craig Tull
- Laurent Vacavant

# Lassi’s Object Networks

- â Colors = data types
- â Modules = behavior
- â Whole network = component
- â Input-output dependency



# Interface Model



# The Module class

â Define a Module class that provides access to the core services:

```
class Module :
       virtual public IModule,
       virtual public IProperty,
       virtual public TEventHandler<SysInitialize>
{
 public:
  IMessageSvc*      msgSvc();
  template< class T>
 StatusCode service(const std::string& name, T*& svc);
    …
}
```

# Setting up - a sample script

â associate States and StateSources

```
StateSource rawFile(inputFile)
next_Record.attach(rawFile)
```

â define Sequences of components to be executed

```
    sequence all =
            { "hitFinder", "tracker", "myanal" }
    sequence reco = { "tracker", "myanal"}
```

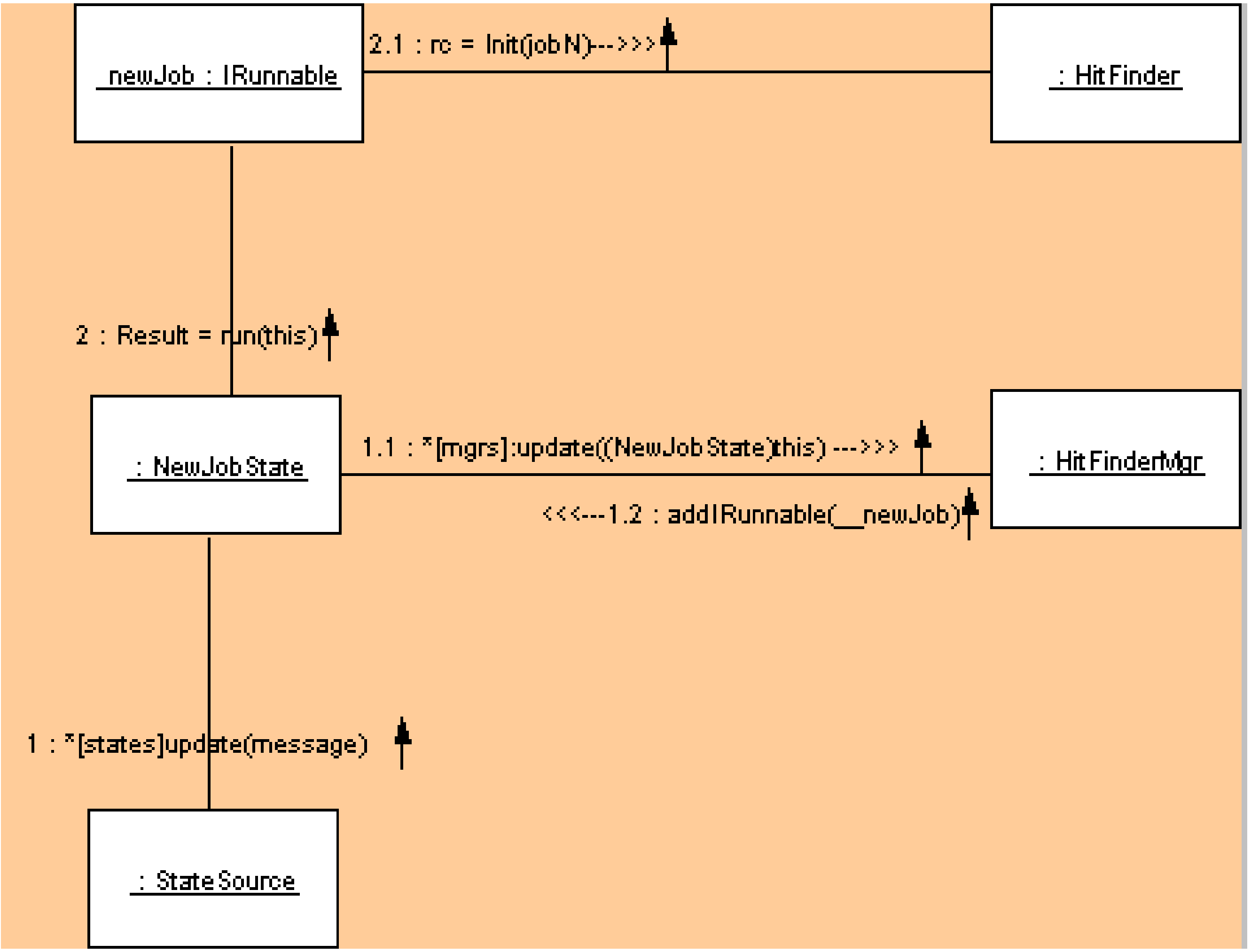
â  define State transitions, with usual flow-control constructs

```
next_run.run("all")
while (next_Record.run("all")) {
          fill_histos.run("reco")
          fill_Bhistos.run("paolo")
}
```

# Running

- â The framework runs States following the script order.
- â Control returns to the framework after each state completes
- â The State tries to run each registered module in order
- â The module determines the status of its associated method, run it if ready, and report to the State.
- â The Object Network (or a Data Manager) notifies modules when their Parameters are ready or change.
- â The State may re-queue a module which is NotReady.

# Scenario: Running a State



newJob : IRunnable

2.1 : rc = Init(jobN)--->>>

: HitFinder

2 : Result = run(this)

: NewJobState

1.1 : *[mgrs]:update((NewJobState)this) --->>>

: HitFinderMgr

<<<---1.2 : addIRunnable(__newJob)

1 : *[states]update(message)

: StateSource

# Scenario: Setting Up

â First we define the state classes

```
DEFINE_CTRL_STATE(NewJobStateS)
DEFINE_CTRL_STATE(NewRunState)
DEFINE_CTRL_STATE(NewRecordState)
```

â Then we create the module managers

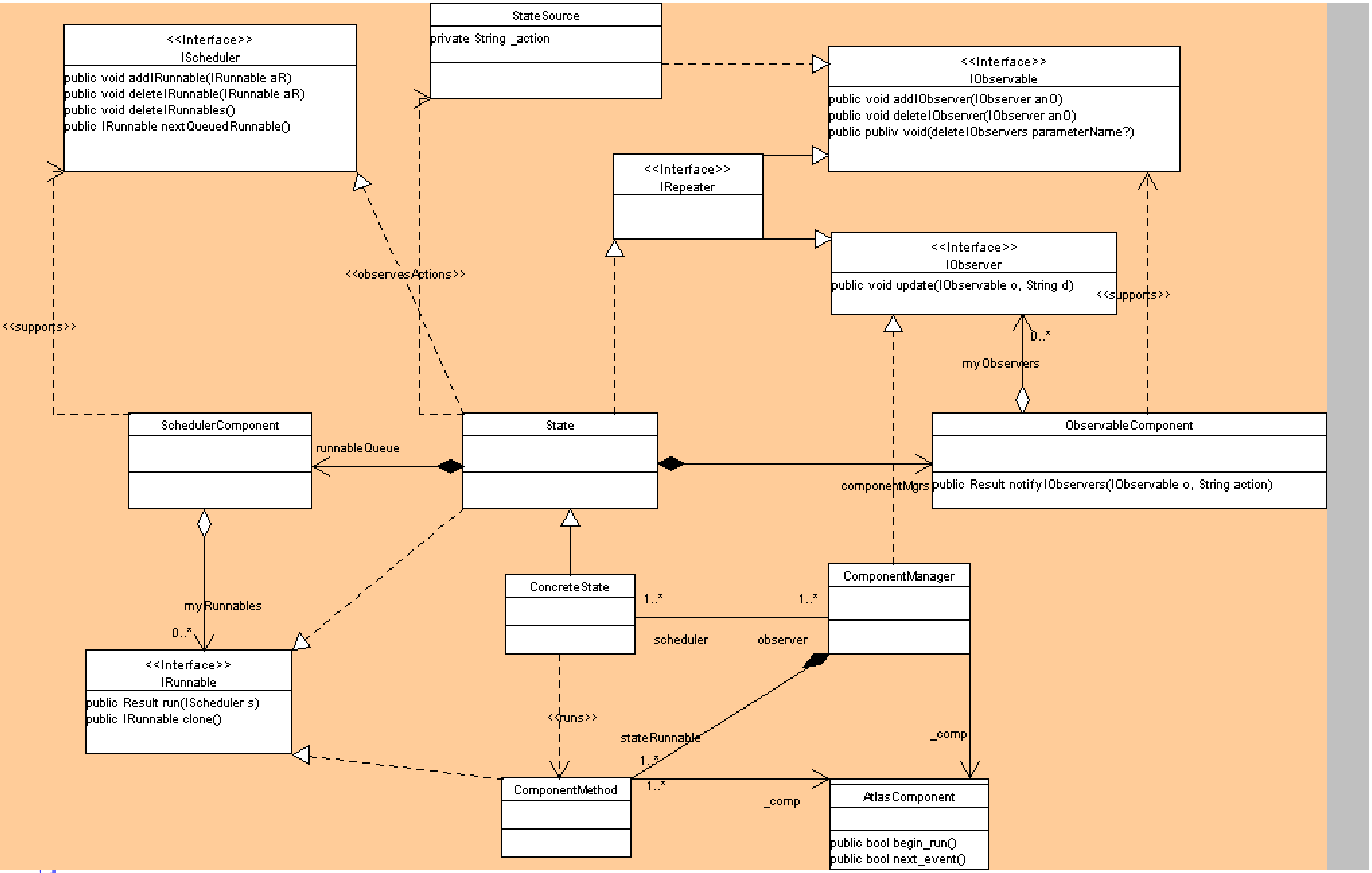
```
HitFinderMgr hitFinder;
HistogrammerMgr myHistos;
```

â We create the states instances and we register the module with them.

```
NewJobState newJob("newJob");
newJob.addIObserver(&myHistos);
newJob.addIObserver(&hitFinder);
```

â Finally we create the state source and register the states with it.

```
StateSource testSource("testSource");
testSource.addIObserver(&newJob);
testSource.addIObserver(&newRun);
testSource.addIObserver(&newRecord);
```



<<Interface>>
IScheduler

public void addIRunnable(IRunnable aR)
public void deleteIRunnable(IRunnable aR)
public void deleteIRunnables()
public IRunnable nextQueuedRunnable()

StateSource

private String _action

<<Interface>>
IObservable

public void addIObserver(IObserver anO)
public void deleteIObserver(IObserver anO)
public publiv void(deleteIObservers parameterName?)

<<Interface>>
IRepeater

<<Interface>>
IObserver

public void update(IObservable o, String d)

<<observesActions>>

<<supports>>

<<supports>>

0..*

myObservers

SchedulerComponent

runnableQueue

State

ObservableComponent

componentMgrs

public Result notifyIObservers(IObservable o, String action)

myRunnables

0..*

ConcreteState

1..*

1..*

ComponentManager

scheduler

observer

<<Interface>>
IRunnable

public Result run(IScheduler s)
public IRunnable clone()

<<runs>>

stateRunnable

1..*

_comp

ComponentMethod

1..*

_comp

AtlasComponent

public bool begin_run()
public bool next_event()