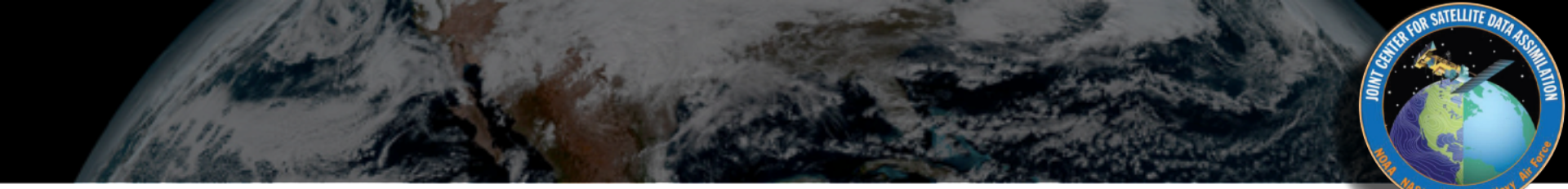


The Joint Effort for Data assimilation Integration

Data Assimilation Structure

Joint Center for Satellite Data Assimilation (JCSDA)

OOPS Approach

Model Space Interfaces

Observation Space Interfaces

Abstract-Concrete Interfaces and Applications

OOPS Analysis and Design

What is data assimilation?
Data assimilation is finding the best estimate (analysis) of the state of the system (atmosphere or other) given a previous estimate of the state (background) and recent observations of the system.

- States properties:
  - Input, output (raw or post-processed).
  - Access values.
  - Move forward in time (using the model).
  - Copy, assign.

- Observations properties:
  - Input, output.
  - Simulate observation from a state (observation operator).
  - Copy, assign.

- We don’t need to know how these operations are performed, how the states are represented or how the observations are stored.

OOPS Analysis and Design

$$J(\mathbf{x}) = \frac{1}{2}(\mathbf{x}_0 - \mathbf{x}_b)^T \mathbf{B}^{-1}(\mathbf{x}_0 - \mathbf{x}_b) + \frac{1}{2}\sum_{i=0}^{n}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]^T \mathbf{R}_i^{-1}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]$$

- Increments:
  - Basic linear algebra operators,
  - Evolve forward in time linearly and backwards with adjoint.
  - Compute as difference between states, add to state.

- Departures:
  - Basic linear algebra operators,
  - Compute as difference between observations, add to observations,
  - Compute as linear variation in observation equivalent as a result of a variation of the state (linearized observation operator).
  - Output (for diagnostics).

- Covariance matrices:
  - Setup,
  - Multiply by matrix (and possibly its inverse).

OOPS Analysis and Design

$$J(\mathbf{x}) = \frac{1}{2}(\mathbf{x}_0 - \mathbf{x}_b)^T \mathbf{B}^{-1}(\mathbf{x}_0 - \mathbf{x}_b) + \frac{1}{2}\sum_{i=0}^{n}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]^T \mathbf{R}_i^{-1}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]$$

- The 4D-Var problem, and the algorithm to solve it, can be described with a very limited number of entities:
  - Vectors: **x**, **y**, **g** and δ**x**.
  - Covariance matrices: **B**, **R** (and eventually **Q**).
  - Two operators and their linearized counterparts: $\mathcal{M}$, **M**, $\mathbf{M}^T$, $\mathcal{H}$, **H**, $\mathbf{H}^T$.
- All data assimilation schemes manipulate the same limited number of entities.
- For future (unknown) developments these entities should be easily available and reusable.
- We have not mentioned any details about how any of the operations are performed, how data is stored or what the model represents.

OOPS Abstract Design

- OOPS is independent of the model and the physical system it represents.
- Flexibility (including yet unknown future development) requires that this goes both ways.
- The components do not know about the high level algorithm currently being run:
  - All actions are driven by the top level code,
  - All data, input and output, is passed by arguments.
- Interfaces must be general enough to cater for all cases, and detailed enough to be able to perform the required actions.
- OOPS stops at the level of the calls to the forecast model and observation operators but the same principle could be applied at any level.

Cost Function Design

- Naive approach:
  - One object for each term of the cost function.
  - Compute each term (or gradient) and add them together.
  - Problem: The model is run several times (Jo, Jc, Jq)

- Another naive approach:
  - Run the model once and store the full 4D state.
  - Compute each term (or gradient) and add them together.
  - Problem: The full 4D state is too big (for us).

- A feasible approach:
  - Run the model once.
  - Compute each term (gradient) as the model is running.
  - Add all the terms together.

Cost Function Implementation

- One class for each term (more flexible).

- Call a method on each object on the fly while the model is running.
  - Uses the PostProcessor structure.
  - Finalize each term and add the terms together at the end.
  - Saving the model linearization trajectory is also the responsibility of a PostProcessor.

- Each formulation derives from an abstract CostFunction base class.
  - Code duplication between strong and weak constraint 4D-Var: use in the same derived class (weak constraint) or write the weak constraint 4D-Var as a sum of strong constraint terms for each sub-window.
  - It was decided to keep 3D-Var and 4D-Var for readability reasons.

- The terms can be re-used (or not), 4D-En-Var was added in a few hours.
  - OO is not magic and will not solve scientific questions by itself.
  - Scientific questions (localization) remain but scientific work can start.
  - Weeks of work would have been necessary in traditional systems.

Joint Effort for Data assimilation Integration

OOPS Approach

Model Space Interfaces

Observation Space Interfaces

Abstract-Concrete Interfaces and Applications

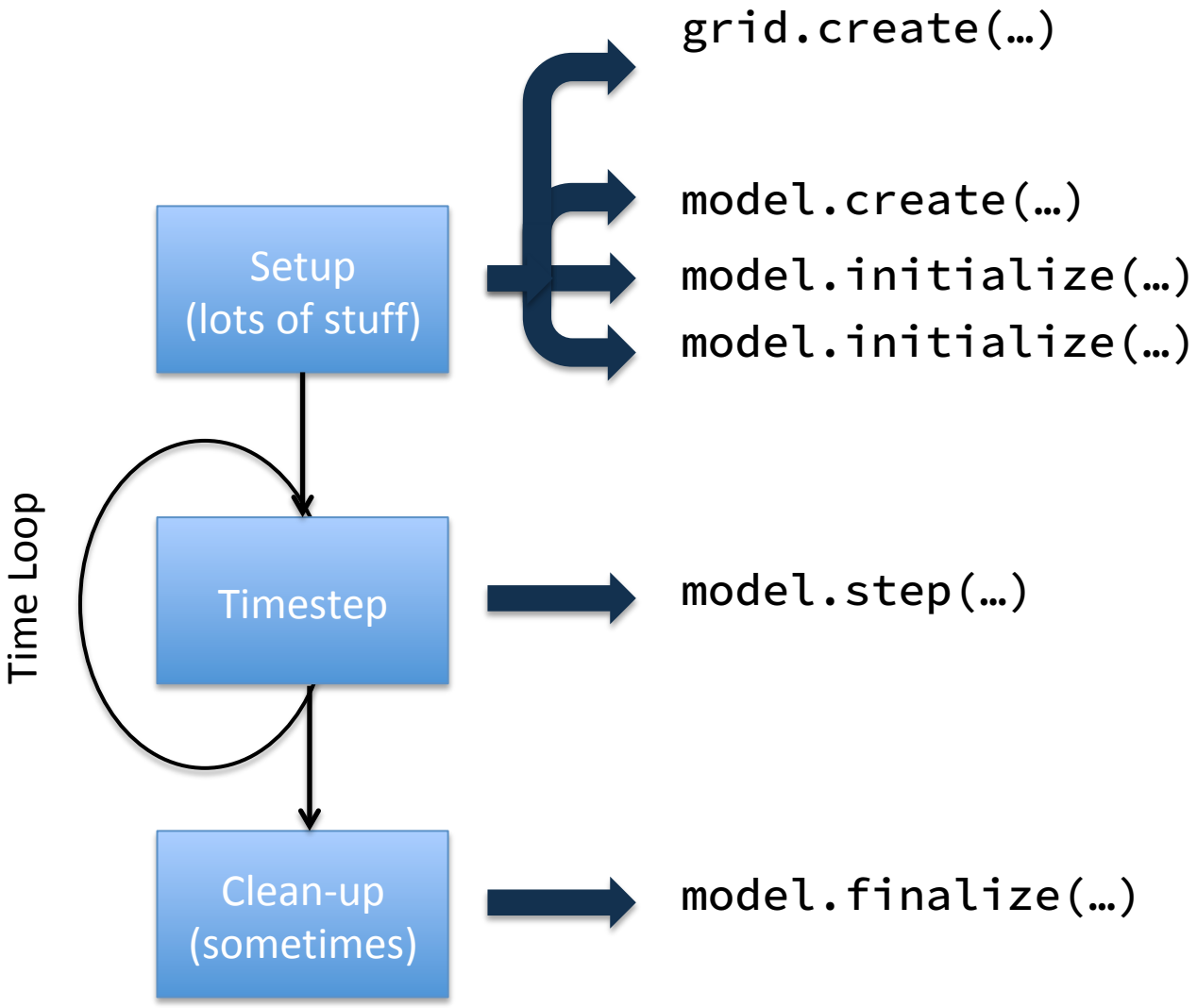
State Class

```
template <typename MODEL>
class State : public util::Printable, private util::ObjectCounter<State<MODEL> > {
public:
  static const std::string classname() {return "oops::State";}

/// Constructor, destructor
  State(const Geometry_ &, const eckit::Configuration &);
  State(const Geometry_ &, const State &);
  State(const State &);
  ~State();
  State & operator=(const State &);

/// Interfacing
  State_ & state() {return *state_;}
  const State_ & state() const {return *state_;}

/// Get state values at observation locations
  void getValues(const Locations_ &, const Variables &, GeoVaLs_ &) const;
  void getValues(const Locations_ &, const Variables &, GeoVaLs_ &, InterpolatorTraj_ &) const;

/// Time
  const util::DateTime validTime() const {return state_->validTime();}

/// I/O and diagnostics
  void read(const eckit::Configuration &);
  void write(const eckit::Configuration &) const;
  double norm() const;
  Geometry_ geometry() const;

private:
  void print(std::ostream &) const;
  boost::scoped_ptr<State_> state_;
};
```

Increment Class

```
template <typename MODEL> class Increment {
 public:
  static const std::string classname() {return "oops::Increment";}

/// Constructor, destructor
  Increment(const Geometry_ &, const Variables &, const util::DateTime &);
  Increment(const Geometry_ &, const Increment &);
  Increment(const Increment &, const bool copy = true);
  virtual ~Increment();

/// Interfacing
  Increment_ & increment() {return *increment_;}
  const Increment_ & increment() const {return *increment_;}

/// Get increment values at observation locations
  void getValuesTL(const Locations_ &, const Variables &, GeoVaLs_ &, const InterpolatorTraj_ &) const;
  void getValuesAD(const Locations_ &, const Variables &, const GeoVaLs_ &, const InterpolatorTraj_ &);

/// Interactions with State
  void diff(const State_ &, const State_ &);

/// Time
  const util::DateTime validTime() const {return increment_->validTime();}
  void updateTime(const util::Duration & dt) {increment_->updateTime(dt);}

/// Linear algebra operators
  void zero();
  void zero(const util::DateTime &);
  Increment & operator =(const Increment &);
  Increment & operator+=(const Increment &);
  Increment & operator-=(const Increment &);
  Increment & operator*=(const double &);
  void axpy(const double &, const Increment &, const bool check = true);
  double dot_product_with(const Increment &) const;
  void schur_product_with(const Increment &);
  void random();
  void accumul(const double &, const State_ &);

/// I/O and diagnostics
  void read(const eckit::Configuration &);
  void write(const eckit::Configuration &) const;

private:
  void print(std::ostream &) const;
  boost::scoped_ptr<Increment_> increment_;
};
```

Model Design: $x_t=M(x_0)$



Model Class

```
template <typename MODEL>
class Model : public util::Printable,
              private boost::noncopyable,
              private util::ObjectCounter<Model<MODEL> >  {
  typedef typename MODEL::Model                  Model_;
  typedef Geometry<MODEL>              Geometry_;
  typedef ModelAuxControl<MODEL>       ModelAux_;
  typedef State<MODEL>                 State_;

 public:
  static const std::string classname() {return "oops::Model";}

  Model(const Geometry_ &, const eckit::Configuration &);
  ~Model();

// Run the forecast
  void forecast(State_ &, const ModelAux_ &,
                const util::Duration &, PostProcessor<State_> &) const;

// Information and diagnostics
  const util::Duration & timeResolution() const {return model_->timeResolution();}

 private:
  void initialize(State_ &) const;
  void step(State_ &, const ModelAux_ &) const;
  void finalize(State_ &) const;
  void print(std::ostream &) const;

  boost::scoped_ptr<Model_> model_;
};
```

Model Class

```
template<typename MODEL>
void Model<MODEL>::forecast(State_ & xx, const ModelAux_ & maux,
                            const util::Duration & len,
                            PostProcessor<State_> & post) const {
  Log::trace() << "Model<MODEL>::forecast starting" << std::endl;
  util::Timer timer(classname(), "forecast");

  const util::DateTime end(xx.validTime() + len);
  this->initialize(xx);
  post.initialize(xx, end, model_->timeResolution());
  post.process(xx);

  while (xx.validTime() < end) {
    this->step(xx, maux);
    post.process(xx);
  }

  post.finalize(xx);
  this->finalize(xx);
  ASSERT(xx.validTime() == end);

  Log::trace() << "Model<MODEL>::forecast done" << std::endl;
}
```

PostProcessors

PostProcessors are called regularly during model integration

- The presence or absence of post processing does not affect the forecast
- PostProcessors isolate the model code from many other unrelated aspects (separation of concerns)

Examples:

- Output of forecast fields
- Generation of products for users
- Computation of digital filter
- Generation of trajectory for linearized models
- Generation of simulated observations
- Diagnostics
- …

LinearModel Class

```
template <typename MODEL>
class LinearModel : public util::Printable,
                    private boost::noncopyable,
                    private util::ObjectCounter<LinearModel<MODEL> > {
 public:
  static const std::string classname() {return "oops::LinearModel";}

  LinearModel(const Geometry_ &, const eckit::Configuration &);
  ~LinearModel();

/// Run the tangent linear forecast
  void forecastTL(Increment_ &, const ModelAuxIncr_ &, const util::Duration &,
                  PostProcessor<Increment_> post = PostProcessor<Increment_>(),
                  PostProcessorTLAD<MODEL> cost = PostProcessorTLAD<MODEL>(),
                  const bool idmodel = false) const;

/// Run the adjoint forecast
  void forecastAD(Increment_ &, ModelAuxIncr_ &, const util::Duration &,
                  PostProcessor<Increment_> post = PostProcessor<Increment_>(),
                  PostProcessorTLAD<MODEL> cost = PostProcessorTLAD<MODEL>(),
                  const bool idmodel = false) const;

// Set the linearization trajectory
  void setTrajectory(const State_ &, State_ &, const ModelAux_ &);

// Information and diagnostics
  const util::Duration & timeResolution() const {return tlm_->timeResolution();}

 protected:
// Run the TL forecast
  void initializeTL(Increment_ &) const;
  void stepTL(Increment_ &, const ModelAuxIncr_ &) const;
  void finalizeTL(Increment_ &) const;

// Run the AD forecast
  void initializeAD(Increment_ &) const;
  void stepAD(Increment_ &, ModelAuxIncr_ &) const;
  void finalizeAD(Increment_ &) const;

 private:
// diagnostics
  void print(std::ostream &) const;

  boost::scoped_ptr<LinearModelBase_> tlm_;
};
```

Model Space Interface Classes

Model space-related classes required by OOPS-JEDI

`Geometry`
`State`
`Increment`
	Geometry, State and Increment

`Model` `LinearModel`	Models
`ErrorCovariance` `(Localization)`	Background error covariance

`ModelAuxControl`
`ModelAuxCovariance`
`ModelAuxIncrement`
	Model bias correction or parameter estimation
	(empty for now)

Joint Effort for Data assimilation Integration

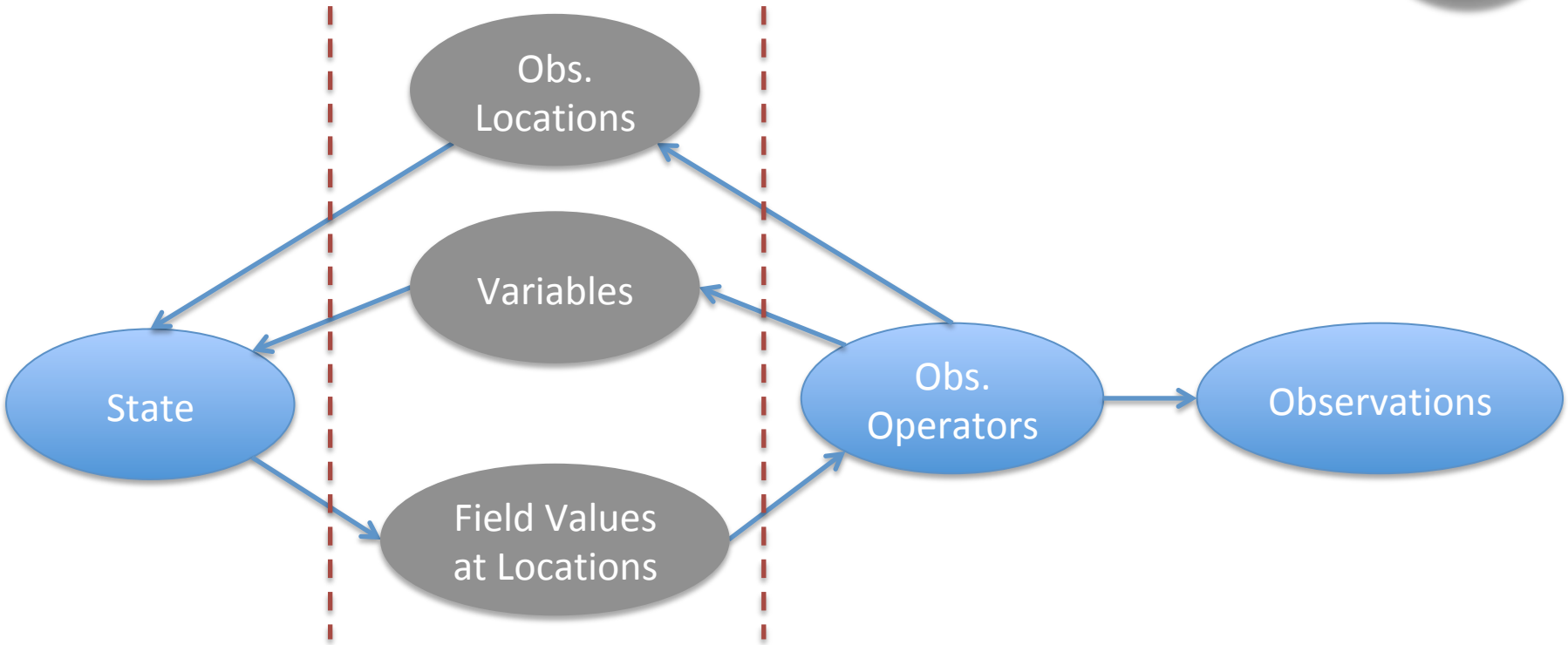
OOPS Approach

Model Space Interfaces

Observation Space Interfaces

Abstract-Concrete Interfaces and Applications

UFO: the interface advantage



- JEDI/UFO introduces standard interfaces between the model and observation worlds
- Observation operators are independent of the model and can easily be shared, exchanged, compared

The Observer

The Observer is a PostProcessor that simulates observations as the model is running

- At each step, the GeoVaLs are populated from the State
- In the finalization, the simulated observations are generated from the GeoVaLs
- The observation filters are applied

```
template <typename MODEL, typename STATE>
void Observer<MODEL, STATE>::doFinalize(const STATE &) {
  Log::trace() << "Observer::doFinalize start" << std::endl;
  for (size_t jj = 0; jj < hop_.size(); ++jj) {
    filters_[jj].priorFilter(*gvals_.at(jj));
    hop_[jj].simulateObs(*gvals_.at(jj), (*yobs_)[jj], ybias_);
    filters_[jj].postFilter((*yobs_)[jj]);
  }
  gvals_.clear();
  Log::trace() << "Observer::doFinalize done" << std::endl;
}
```

Observations Class

Observations are organized by type, although types can be any convenient grouping

```
template <typename MODEL> class Observations : public util::Printable {
 public:
  explicit Observations(const ObsSpace_ &);
  explicit Observations(const Observations &);
  ~Observations();
  Observations & operator=(const Observations &);

/// Access
  std::size_t size() const {return obs_.size();}
  ObsVector_ & operator[](const std::size_t ii) {return obs_.at(ii);}
  const ObsVector_ & operator[](const std::size_t ii) const {return obs_.at(ii);}

/// Interactions with Departures
  std::vector<boost::shared_ptr<ObsVector_> > operator-(const Observations & other) const;
  Observations & operator+=(const Departures_ &);

/// Save observations values
  void save(const std::string &) const;
  void read(const eckit::Configuration &);

 private:
  void print(std::ostream &) const;

  boost::ptr_vector<ObsVector_> obs_;
};
```

Departures Class

```
class Departures : public util::Printable, public GeneralizedDepartures {
 public:
// Constructors and destructor
  explicit Departures(const ObsSpace_ &);
  explicit Departures(std::vector<boost::shared_ptr<ObsVector_> >);
  explicit Departures(const Departures &);
  ~Departures();

/// Access
  std::size_t size() const {return dep_.size();}
  ObsVector_ & operator[](const std::size_t ii) {return *dep_.at(ii);}
  const ObsVector_ & operator[](const std::size_t ii) const {return *dep_.at(ii);}

// Linear algebra operators
  Departures & operator=(const Departures &);
  Departures & operator+=(const Departures &);
  Departures & operator-=(const Departures &);
  Departures & operator*=(const double &);
  Departures & operator*=(const Departures &);
  Departures & operator/=(const Departures &);
  void zero();
  void invert();
  void axpy(const double &, const Departures &);
  double dot_product_with(const Departures &) const;

/// Save departures values
  void save(const std::string &) const;

 private:
  void print(std::ostream &) const;
  std::vector<boost::shared_ptr<ObsVector_> > dep_;
};
```

ObsVector Class

```
template <typename MODEL>
class ObsVector : public util::Printable,
                  private util::ObjectCounter<ObsVector<MODEL> > {
 public:
  explicit ObsVector(const ObservationSpace<MODEL> &);
  explicit ObsVector(const ObsVector &, const bool copy = true);
  explicit ObsVector(ObsVector_ *);
  ~ObsVector();
/// Interfacing
  ObsVector_ & obsvector() {return *data_;}
  const ObsVector_ & obsvector() const {return *data_;}
// Linear algebra
  ObsVector & operator = (const ObsVector &);
  ObsVector & operator*= (const double &);
  ObsVector & operator+= (const ObsVector &);
  ObsVector & operator-= (const ObsVector &);
  ObsVector & operator*= (const ObsVector &);
  ObsVector & operator/= (const ObsVector &);
  void zero();
  void axpy(const double &, const ObsVector &);
  void invert();
  void random();
  double dot_product_with(const ObsVector &) const;
  unsigned int size() const {return data_->size();}
// I/O
  void read(const std::string &);
  void save(const std::string &) const;
 private:
  void print(std::ostream &) const;
  boost::scoped_ptr<ObsVector_> data_;
};
```

ObsOperator Class

```
template <typename MODEL>
class ObsOperator : public util::Printable,
                    private boost::noncopyable,
                    private util::ObjectCounter<ObsOperator<MODEL> > {
 public:
  static const std::string classname() {return "oops::ObsOperator";}

  explicit ObsOperator(const ObsSpace_ &);
  ~ObsOperator();

/// Obs Operator
  void simulateObs(const GeoVaLs_ &, ObsVector_ &, const ObsAuxControl_ &) const;

/// Interfacing
  const ObsOperatorBase_ & obsoperator() const {return *oper_;}

/// Other
  const Variables & variables() const;  // Required inputs variables from Model

 private:
  void print(std::ostream &) const;
  boost::scoped_ptr<ObsOperatorBase_> oper_;
};
```

LinearObsOperator Class

```
template <typename MODEL>
class LinearObsOperator : public util::Printable,
                          private boost::noncopyable,
                          private util::ObjectCounter<LinearObsOperator<MODEL> > {
public:
  static const std::string classname() {return "oops::LinearObsOperator";}

  explicit LinearObsOperator(const ObsSpace_ &);
  ~LinearObsOperator();

/// Interfacing
  const LinearObsOperBase_ & linearobsoperator() const {return *oper_;}

/// Obs Operators
  void setTrajectory(const GeoVaLs_ &, const ObsAuxControl_ &);
  void obsEquivTL(const GeoVaLs_ &, ObsVector_ &, const ObsAuxIncrement_ &) const;
  void obsEquivAD(GeoVaLs_ &, const ObsVector_ &, ObsAuxIncrement_ &) const;

/// Other
  const Variables & variables() const;  // Required inputs variables from LinearModel

 private:
  void print(std::ostream &) const;
  boost::scoped_ptr<LinearObsOperBase_> oper_;
};
```

Observation Interface Classes

Observation-related classes implemented in OOPS-JEDI

`Observations` `Departures`	Next slides

Observation-related classes required by OOPS-JEDI

`GeoVaLs`
`Locations`
`Variables`
	UFO/OOPS interfaces

`ObservationSpace` `ObsVector`	IODA

`ObsOperator`
`LinearObsOperator`
`ObsErrorCovariance`
	UFO

`ObsAuxControl`
`ObsAuxIncrement`
`ObsAuxCovariance`
	For bias correction (not implemented yet in UFO)

Joint Effort for Data assimilation Integration

OOPS Approach

Model Space Interfaces

Observation Space Interfaces

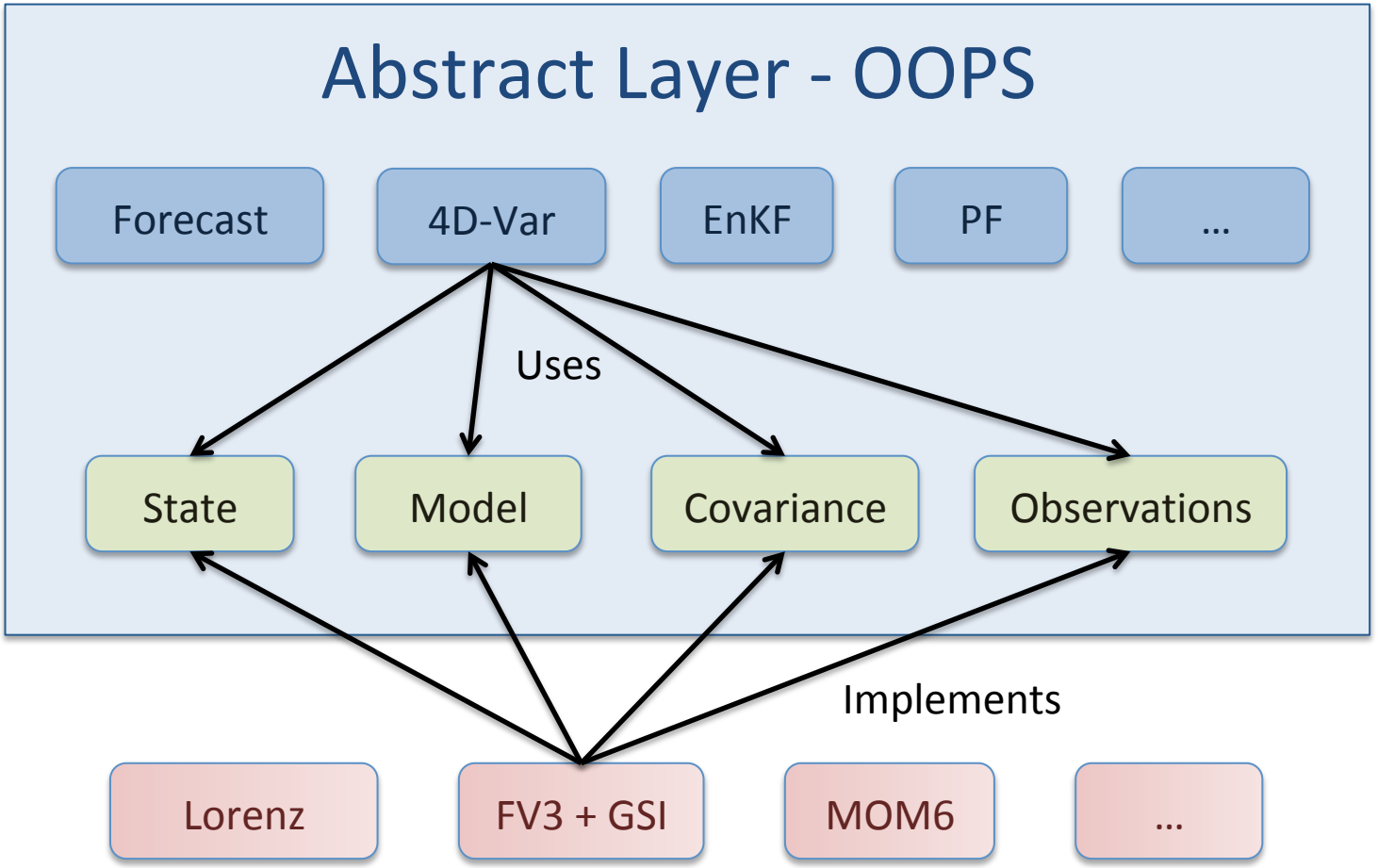
Abstract-Concrete Interfaces and Applications

Design: separation of concerns

Generic Applications

Abstract building blocks

Systems



Abstract interfaces are the most important aspect of the design

Interfaces Classes

All model or observation specific classes are wrapped into an interface class

- Using the C++ templates this is not strictly necessary but provides a convenient place to group all interfaces and make them visible

Each interface class contains a (smart) pointer to the actual implementation defined by the “model trait” and passes the calls down

The interface layer is also used to instrument the code

- Timing statistics
- Trace execution

Interface classes are all in `oops/src/oops/interface`

Interfaces Classes

```
template <typename MODEL>
class State {
  typedef typename MODEL::State      State_;
 public:
/// Most methods have been removed for readability

/// Interfacing
  State_ & state() {return *state_;}
  const State_ & state() const {return *state_;}

/// Get state values at observation locations
  void getValues(const Locations_ &, const Variables &,
                 GeoVaLs_ &) const;

 private:
  boost::scoped_ptr<State_> state_;
};
```

Class of specific implementation (from trait)

Access specific object (only for use in interface classes)
Name of method is name of class in lowercase

Smart pointer to actual object

Interfaces Classes

Example of a method in an interface class

```
template<typename MODEL>
void State<MODEL>::getValues(const Locations_ & locs,
                             const Variables & vars,
                             GeoVaLs_ & gvals) const {
  Log::trace() << "State<MODEL>::getValues starting" << std::endl;
  util::Timer timer(classname(), "getValues");
  state_->getValues(locs.locations(), vars, gvals.geovals());
  Log::trace() << "State<MODEL>::getValues done" << std::endl;
}
```

Trace method on entry and exit

Timer will be destructed when going out of scope Constructor and destructor do the work

Method of specific implementation is called, with actual arguments

JEDI: Main programs

Standard C++ main

Run object for technical setup

```
int main(int argc,  char ** argv) {
  oops::Run run(argc, argv);
  oops::Variational<qg::QgTraits> var;
  run.execute(var);
  return 0;
}
```

Create Application object

Execute Application

Done

JEDI: Applications

The Application for variational DA is in
`src/oops/runs/Variational.h`

The Variational class inherits from Application (is an application)

Applications are the normal mode of execution

Unit tests are specific applications that run tests for a given class
- The main program manages tests and adds test reporting
- This is hidden for most users and developers

All OOPS-JEDI executables take one argument:
- a configuration file in json/yaml format