OOPS Model Space

Jedi Academy IV, Monterey CA

26th February 2020

U.S. AIR FORCE

Introduction

- OOPS consists of a number of generic classes, each designed to handle a specific task or set of tasks without knowing the kind of model for which data assimilation is being performed.

- Some of these classes require a specific implementation in order to perform their task. Consider the state or observation operator for example. There are classes to handle these in oops but they do little more than specify the interfaces without a specific state or observation operator.

- Other classes can have meaning without a specific implementation, e.g. the CostFunction class. However these classes inevitably make use of other classes with a specific implementation. For example a cost function uses, among other things, the state class.

- In JEDI we refer to the classes that require specific implementation as interface classes. The collection of these implemented classes dependent on the forecast model is referred to as the ***Model Space.***

Interfaces to templates

Everywhere in the OOPS code are lines like this that proceed classes, methods, types etc.

```
template <typename MODEL>
```

This templating (described in other lectures) lets the classes or methods behave a certain way, for a certain model. MODEL contains what we call model space (and observation space) and is a list of the ways the interface classes should be templated.

Interface class

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

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

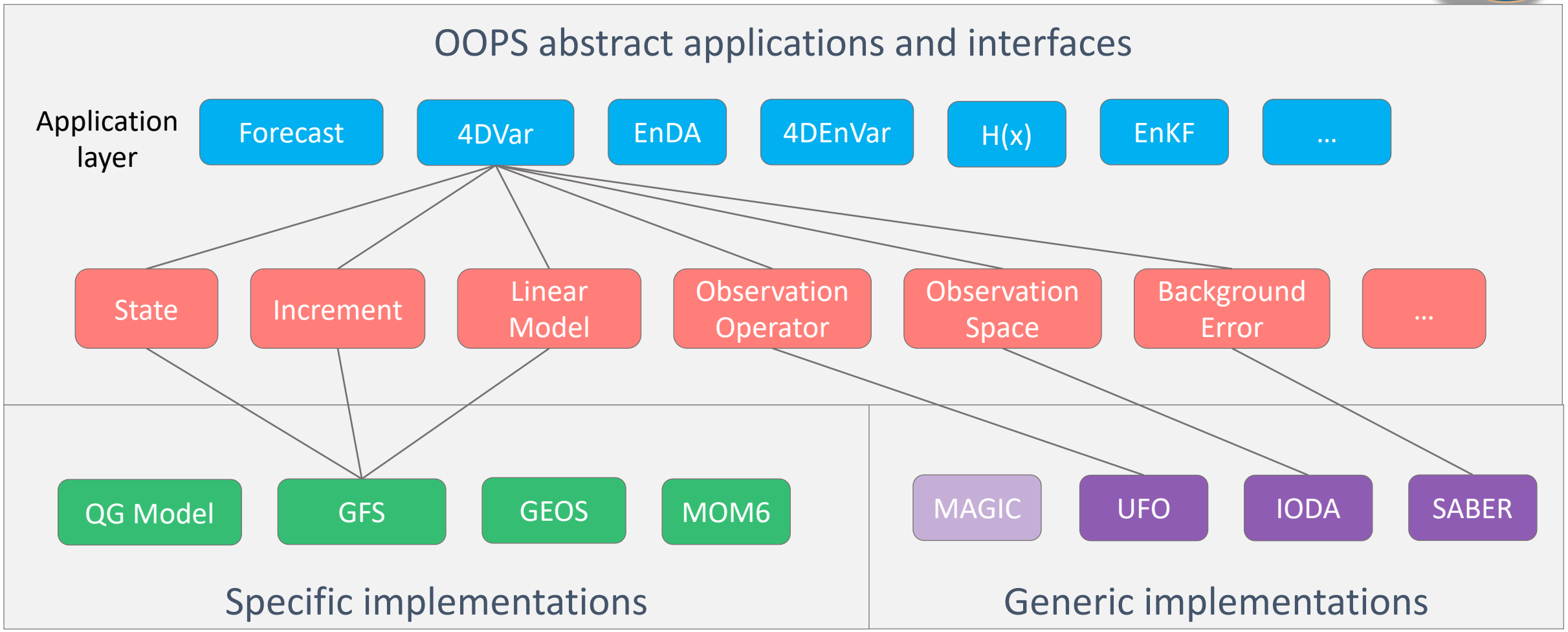
Each interface class contains a pointer to the actual implementation defined by the “model trait” and essentially just passes the calls down

The interface layer is also used to instrument the code

- Timing statistics
- Trace execution

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

The power of interface classes



Implementations don't know which applications they are part of and the applications don't know which model is being used.

The Interface Classes

Dependent on Forecast model

- ErrorCovariance
- Geometry
- GetValues
- Increment
- LinearModel
- LinearVariableChange
- Localization
- Model
- ModelAuxControl
- ModelAuxCovariance
- ModelAuxIncrement
- State
- VariableChange

Dependeny on observations (UFO/IODA)

- GeoVals
- LinearObsOperator
- Locations
- ObsAuxControl
- ObsAuxCovariance
- ObsAuxIncrement
- ObsErrorCovariance
- ObservationSpace
- ObsOpertor
- ObsVector

Hybrid 4DVar application

Incremental hybrid-4DVar involves a number of linear and nonlinear variable transforms:

$$\frac{\partial J}{\partial \delta \mathbf{x}_0} = \mathbf{B}^{-1}\left(\delta \mathbf{x}_0 - \delta \mathbf{x}_b\right) - \sum_{k=0}^{K} \mathbf{K}_M^{\top} \mathbf{M}_k^{\top} \mathbf{K}_h^{\top} \mathbf{H}^{\top} \mathbf{R}_k^{-1}\left(\mathbf{d}_k - \mathbf{H}\mathbf{K}_h \delta \mathbf{x}_k\right)$$

LinearModel

$$\delta \mathbf{x}_k = \mathbf{M}_{t_{k-1} \rightarrow t_k} \mathbf{M}_{t_{k-1} \rightarrow t_{k-2}} \cdots \mathbf{M}_{t_0 \rightarrow t_1} \mathbf{K}_m \delta \mathbf{x}_0$$

Increment

$$\mathbf{d}_k = \mathbf{y}_k^o - h\left(k_h\left\{m_{t_0 \rightarrow t_k}\left[k_m\left(\mathbf{x}_0\right)\right]\right\}\right)$$

LinearVariableChange

ErrorCovariance

$$\mathbf{B} = \mathbf{K}_b \mathbf{D}\mathbf{C}\mathbf{D}\mathbf{K}_b^{\top} + \mathbf{L} \circ \mathbf{X}\mathbf{X}^{\top}$$

State

Localization

Interface class - specification

```
// ----------------------------------------------------------------------

template <typename MODEL>
class Geometry  {
  typedef typename MODEL::Geometry Geometry_;

 public:

  Geometry(const eckit::Configuration &, const eckit::mpi::Comm &);
  Geometry(const Geometry &);
  explicit Geometry(boost::shared_ptr<const Geometry_>);
  ~Geometry();

  /// Interfacing
  const Geometry_ & geometry() const {return *geom_;}

 private:
  boost::shared_ptr<const Geometry_> geom_;
};

// ----------------------------------------------------------------------
```

Class of specific implementation (from trait)

Defines interfaces of methods

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

Shared pointer to actual object

Interface class - method

Example of a method in an interface class

```
// -------------------------------------------------------------------------

template <typename MODEL>
Geometry<MODEL>::Geometry(const eckit::Configuration & conf,
                          const eckit::mpi::Comm & comm): geom_() {
  Log::trace() << "Geometry<MODEL>::Geometry starting" << std::endl;
  util::Timer timer(classname(), "Geometry");
  geom_.reset(new Geometry_(conf, comm));
  Log::trace() << "Geometry<MODEL>::Geometry 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

Passing the Model Space

TRAITS

```
struct Traits {
   typedef myModel::Geometry Geometry;
}
```

DRIVER

```
#include traits.h
int main(int argc, char ** argv) {
  oops::SomeApplication<Traits> app;
  app.execute(Config)
}
```

APPLICATION

```
template <typename MODEL> class SomeApplication {
  int execute(const eckit::Configuration & Config) const {
    const Geometry<MODEL> resol(Config);
  }
```

INTERFACE

```
template <typename MODEL>
class Geometry {

 public:
  explicit Geometry(const eckit::Configuration &);
  ...
 private:
   boost::shared_ptr<const typename MODEL::Geometry> geom_;
};
```

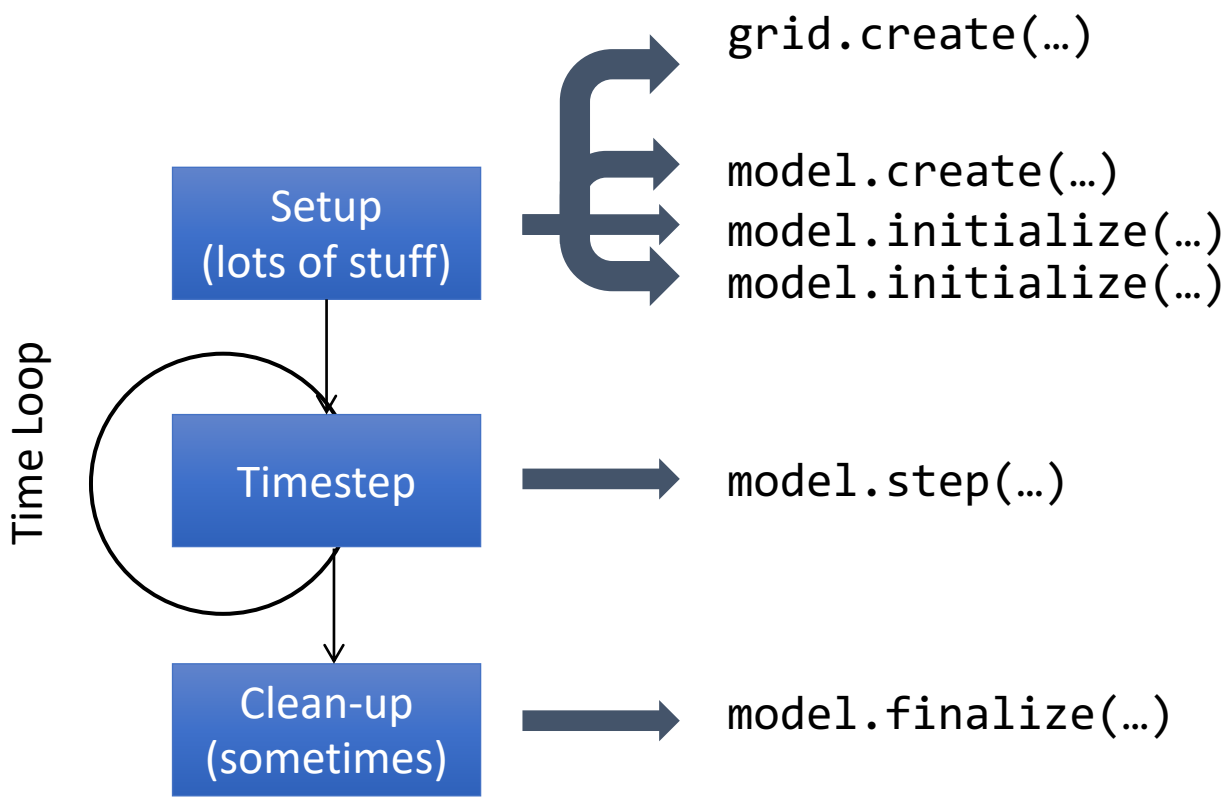
Model implements a Geometry class

Top level main creates an application object passing the traits for the templating.

The application passes the traits down creating the interface objects it needs.

The interface class creates the concrete object from the traits.

Model design: $x_t=M(x_0)$

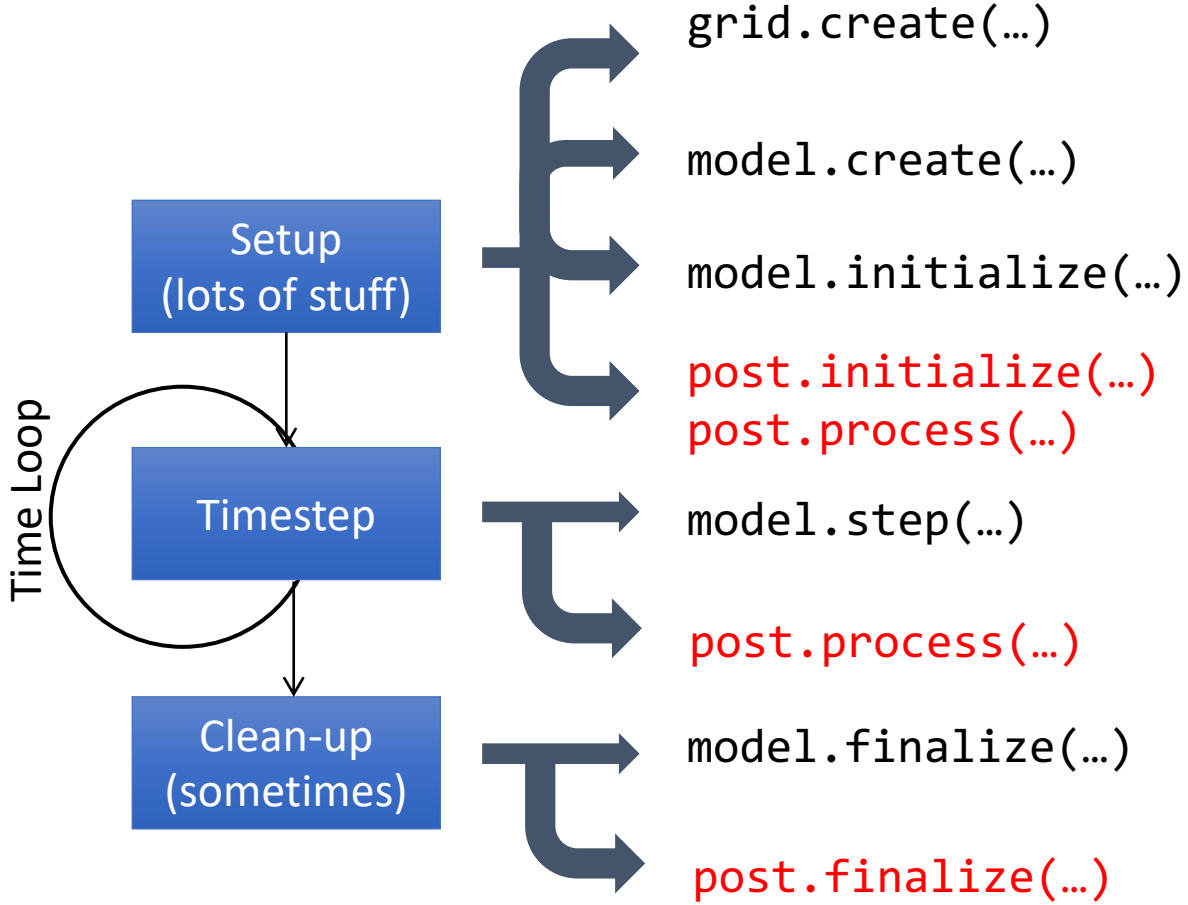


Model design: post processing

The 4D model state is never stored in memory. Post processors are called that have access to the model state but then the model continues and the intermediate states are not stored.

Examples of post processors:

- Calling the observation operator.
- Saving the (low res) trajectory for the TLM and adjoint.
- Writing output to file.
- Printing state information to a stream.



Geometry class

```
template <typename MODEL>
class Geometry : public util::Printable,
                 private util::ObjectCounter<Geometry<MODEL> > {
  typedef typename MODEL::Geometry              Geometry_;
  typedef GeometryIterator<MODEL>               GeometryIterator_;

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

  explicit Geometry(const eckit::Configuration &);
  Geometry(const Geometry &);
  explicit Geometry(boost::shared_ptr<const Geometry_>);
  ~Geometry();

/// Interfacing
  const Geometry_ & geometry() const {return *geom_;}

  GeometryIterator_ begin() const;
  GeometryIterator_ end()   const;

 private:
  Geometry & operator=(const Geometry &);
  void print(std::ostream &) const;
  boost::shared_ptr<const Geometry_> geom_;
};
```

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 Variables, const util::dateTime &);
  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_;}

/// 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;

  void accumul(const double&, const state&)

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

Increment class

```
template <typename MODEL>
class Increment : public oops::GeneralizedDepartures,
                  public util::Printable,
                  private util::ObjectCounter<Increment<MODEL> > {
  typedef typename MODEL::Increment  Increment_;
  typedef Geometry<MODEL>            Geometry_;
  typedef GeoVaLs<MODEL>             GeoVaLs_;
  typedef InterpolatorTraj<MODEL>    InterpolatorTraj_;
  typedef Locations<MODEL>           Locations_;
  typedef State<MODEL>               State_;

 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

/// 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 &);
  void dirac(const eckit::Configuration &);
  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;
  double norm() const;

/// Get geometry
  Geometry_ geometry() const;

/// Unstructured grid
  void ug_coord(UnstructuredGrid &) const;
  void field_to_ug(UnstructuredGrid &, const int & = 0) const;
  void field_from_ug(const UnstructuredGrid &, const int & = 0);

/// Serialize and deserialize
  size_t serialSize() const;
  void serialize(std::vector<double> &) const;
  void deserialize(const std::vector<double> &, size_t &);

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

GetValues

In order to maintain the separation of concerns the observation operator is split into a model dependent parts and model agnostic part.

$$y^o = h(x)$$
$$= h_{obs}\left[h_{mod}(x)\right]$$

The model dependent part might involve interpolation, field of view calculations and variable transforms.

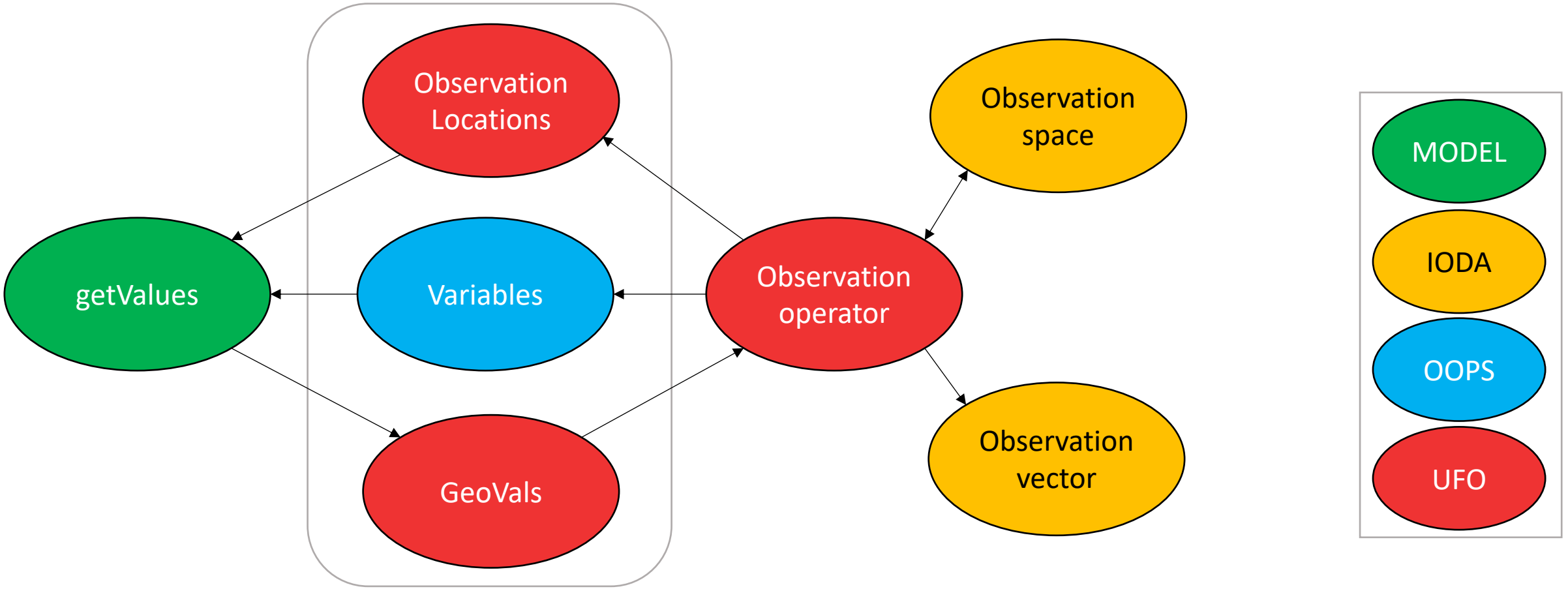
The intermediate state after computing the model dependent part of the observation operator are known as GeoVaLs (Geophysical Values at observation Locations).

$$GeoVaLs = h_{mod}(x)$$

These are model states interpolated to observation locations and converted to the variables requested by the observation operator.

OOPS – UFO – IODA – MODEL: the interface advantage

- JEDI/UFO introduces standard interfaces between the model and observation worlds.
- Observation operators are independent of the model, easy sharing, exchange and comparison.



GetValues

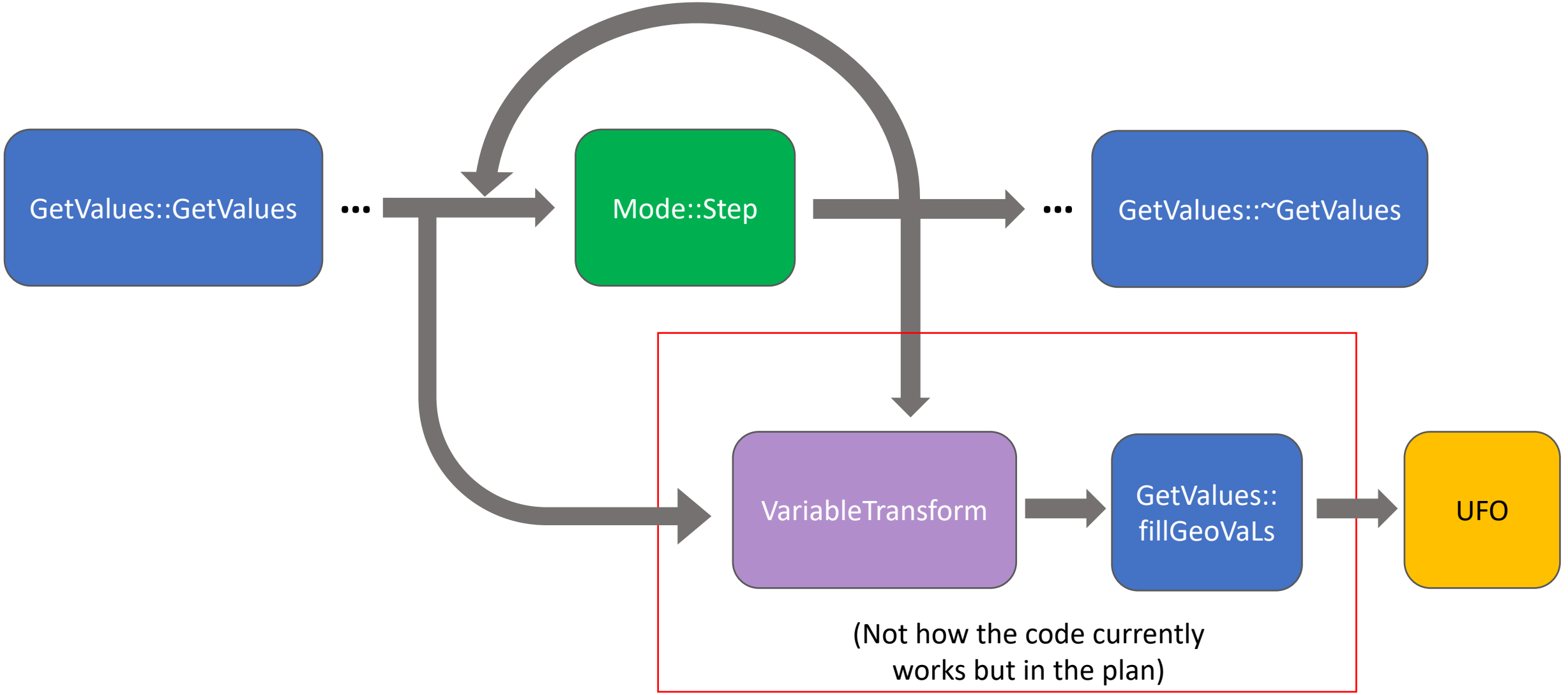
```
template <typename MODEL>
class GetValues : public util::Printable,
                  private util::ObjectCounter<GetValues<MODEL> > {
  typedef typename MODEL::GetValues  GetValues_;
  typedef Geometry<MODEL>            Geometry_;
  typedef GeoVaLs<MODEL>             GeoVaLs_;
  typedef Locations<MODEL>           Locations_;
  typedef State<MODEL>               State_;

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

/// Constructor, destructor
  GetValues(const Geometry_ &, const Variables &, const Locations_ &);
  ~GetValues();

/// Interfacing
  GetValues_ & getvalues() {return *getvalues_;}
  const GetValues_ & getvalues() const {return *getvalues_;}

/// Get state values at observation locations
  void fillGeoVaLs(const State_ &, const util::DateTime &, const util::DateTime &,
                   GeoVaLs_ &) const;

 private:
  void print(std::ostream &) const;
  std::unique_ptr<GetValues_> getvalues_;
};
```

Called once per outer loop

Called every time step

Note that this is a branch new class, still in review. GetValues is moving from the state and increment.

GetValues


GetValues::GetValues
Mode::Step
GetValues::~GetValues
VariableTransform
GetValues::
fillGeoVaLs
UFO
(Not how the code currently
works but in the plan)

LinearGetValues

```
template <typename MODEL>
class LinearGetValues : public util::Printable,
                        private util::ObjectCounter<LinearGetValues<MODEL> > {
  typedef typename MODEL::LinearGetValues  LinearGetValues_;
  typedef Geometry<MODEL>            Geometry_;
  typedef GeoVaLs<MODEL>             GeoVaLs_;
  typedef Increment<MODEL>           Increment_;
  typedef Locations<MODEL>           Locations_;
  typedef State<MODEL>               State_;

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

/// Constructor, destructor
  LinearGetValues(const Geometry_ &, const Variables &, const Locations_ &);
  virtual ~LinearGetValues();

/// Interfacing
  LinearGetValues_ & lingetvalues() {return *lingetvalues_;}
  const LinearGetValues_ & lingetvalues() const {return *lingetvalues_;}

  void setTrajectory(const State_ &, const util::DateTime &, const util::DateTime &,
                     GeoVaLs_ &);
/// Get lingetvalues values at observation locations
  void computeTL(const Increment_ &, const util::DateTime &, const util::DateTime &,
                 GeoVaLs_ &) const;
  void computeAD(Increment_ &, const util::DateTime &, const util::DateTime &,
                 const GeoVaLs_ &) const;

 private:
  void print(std::ostream &) const;
  std::unique_ptr<LinearGetValues_> lingetvalues_;
};
```

Called once per outer loop

Called every nonlinear time step

Called every time step every inner loop

Model class

```
template <typename MODEL>
class Model : public util::Printable,
              private boost::noncopyable,
              private util::ObjectCounter<Model<MODEL> >  {
  typedef ModelBase<MODEL>           ModelBase_;
  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 &);
  virtual ~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();}
  const oops::Variables & variables() const {return model_->variables();}

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

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

Model class | forecast

```
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);
  Log::info() << "Model:forecast: forecast starting: " << xx << std::endl;
  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);
  Log::info() << "Model:forecast: forecast finished: " << xx << std::endl;
  ASSERT(xx.validTime() == end);

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

ModelBase Factory

Some of the constructors in JEDI use factories. This is a powerful way of constructing objects in JEDI that allows for run time choice of constructor without lots of messy if statements.

The Model class in JEDI is an example of using a factory.

```
template <typename MODEL>
class ModelFactory {
  typedef Geometry<MODEL>   Geometry_;
 public:
  static ModelBase<MODEL> * create(const Geometry_ &, const eckit::Configuration &);
  virtual ~ModelFactory() {}
 protected:
  explicit ModelFactory(const std::string &);
 private:
  virtual ModelBase<MODEL> * make(const Geometry_ &, const eckit::Configuration &) = 0;
  static std::map < std::string, ModelFactory<MODEL> * > & getMakers() {
    static std::map < std::string, ModelFactory<MODEL> * > makers_;
    return makers_;
  }
};
```

Pseudo model

Four dimensional data assimilation algorithms require forecasts through the data assimilation window.

Typically this is done by connecting JEDI to the forecast model and stepping the model in time while calling the post processors.

Alternatively the forecast can be achieved using IO. In OOPS there is a generic pseudo model class in development.

Instead of the step method containing a call to the forecast model is call the state.read method and reads a model state from a previously run forecast.

LinearModel class

```
template <typename MODEL>
class LinearModel : public util::Printable,
                    private boost::noncopyable,
                    private util::ObjectCounter<LinearModel<MODEL> >  {
  typedef LinearModelBase<MODEL>     LinearModelBase_;
  typedef Increment<MODEL>           Increment_;
  typedef Geometry<MODEL>            Geometry_;
  typedef ModelAuxControl<MODEL>     ModelAux_;
  typedef ModelAuxIncrement<MODEL>   ModelAuxIncr_;
  typedef State<MODEL>               State_;

 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();}
  const oops::Variables & variables() const {return tlm_->variables();}

 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;

  std::unique_ptr<LinearModelBase_> tlm_;
};
```

ErrorCovariance Class

```
template <typename MODEL>
class ErrorCovariance : public oops::ModelSpaceCovarianceBase<MODEL>,
                        public util::Printable,
                        private util::ObjectCounter<ErrorCovariance<MODEL> >,
                        private boost::noncopyable {
  typedef typename MODEL::Covariance Covariance_;
  typedef Geometry<MODEL>            Geometry_;
  typedef Increment<MODEL>           Increment_;
  typedef State<MODEL>               State_;

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

  ErrorCovariance(const Geometry_ &, const Variables &, const eckit::Configuration &,
                  const State_ &, const State_ &);
  virtual ~ErrorCovariance();

 private:
  void doRandomize(Increment_ &) const override;
  void doMultiply(const Increment_ &, Increment_ &) const override;
  void doInverseMultiply(const Increment_ &, Increment_ &) const override;

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

  boost::scoped_ptr<Covariance_> covariance_;
};
```

VariableChange class

```
template <typename MODEL, typename CHVAR>
class VariableChange : public oops::VariableChangeBase<MODEL> {
  typedef Geometry<MODEL>             Geometry_;
  typedef State<MODEL>                State_;

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

  VariableChange(const Geometry_ &, const eckit::Configuration &);
  virtual ~VariableChange();

  void changeVar(const State_ &, State_ &) const override;
  void changeVarInverse(const State_ &, State_ &) const override;

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

  boost::scoped_ptr<CHVAR> chvar_;
};
```

LinearVariableChange

```
template <typename MODEL, typename CHVAR>
class LinearVariableChange : public oops::LinearVariableChangeBase<MODEL> {
  typedef Geometry<MODEL>              Geometry_;
  typedef Increment<MODEL>             Increment_;
  typedef State<MODEL>                 State_;

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

  LinearVariableChange(const State_ &, const State_ &,
                       const Geometry_ &, const eckit::Configuration &);
  virtual ~LinearVariableChange();

  void multiply(const Increment_ &, Increment_ &) const override;
  void multiplyInverse(const Increment_ &, Increment_ &) const override;
  void multiplyAD(const Increment_ &, Increment_ &) const override;
  void multiplyInverseAD(const Increment_ &, Increment_ &) const override;

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

  boost::scoped_ptr<CHVAR> chvar_;
};
```