Joint Effort for Data assimilation Integration

Unified Forward Operator Code Sprint

Yannick Trémolet

JCSDA

6 November 2017

Modular

- The weather forecasting problem can be broken into manageable pieces:
  - Data assimilation (or ensemble prediction) can be described without knowing the specifics of a model or observations.
  - Minimisation algorithms can be written without knowing the details of the matrices and vectors involved.
  - Development of a dynamical core on a new model grid should not require knowledge of the data assimilation algorithm.

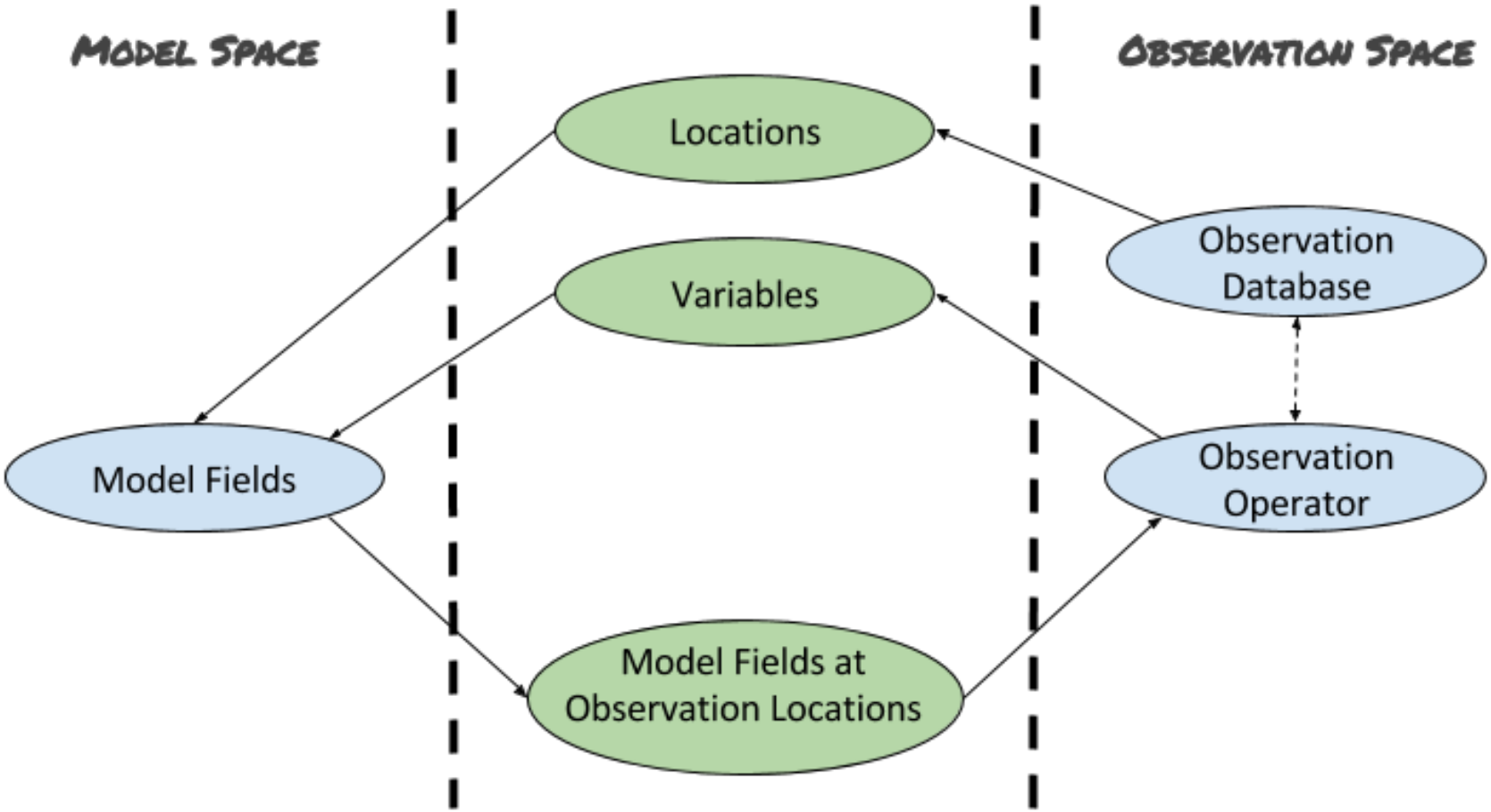
- Unfortunately, in most cases, Fortran modules don't lead to modular codes.
  - Very few codes use Fortran modules as more than glorified common blocks.

- We need to go beyond modules: classes and object oriented programming.

Separation of concerns

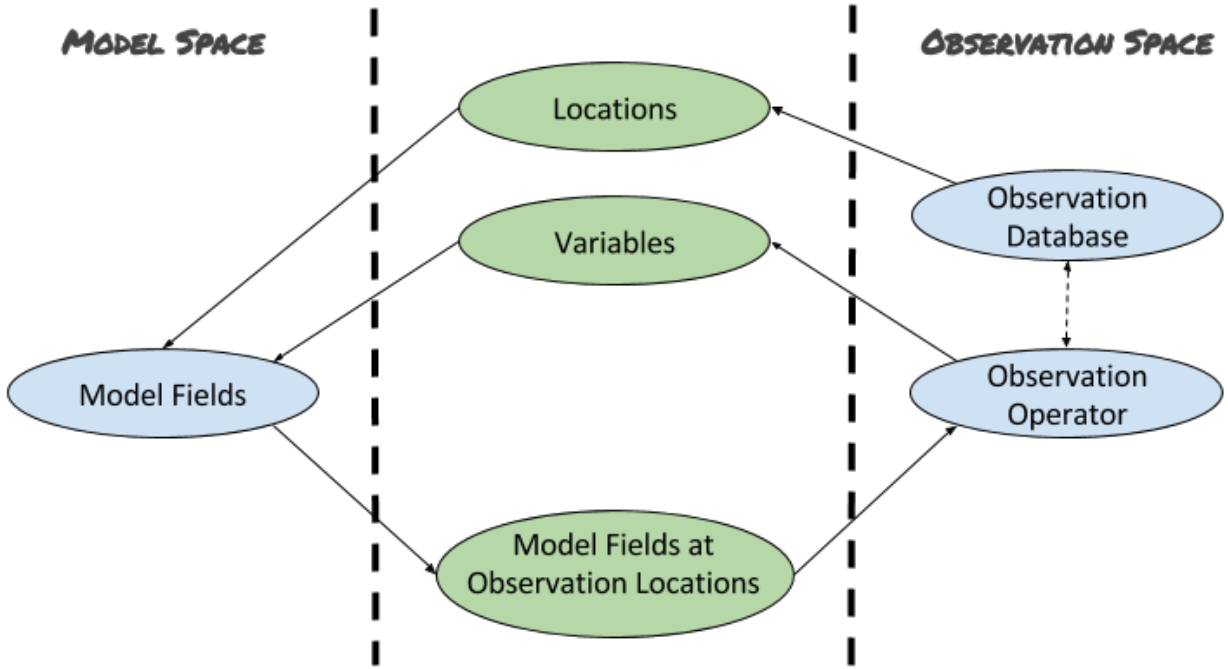
- All aspects exist but scientists focus on one aspect at a time.
- Different concepts should be treated in different parts of the code that interact through interfaces.
- Each class has clearly defined and limited responsibilities.

Generic UFO



- JEDI/UFO introduce standard interfaces between models and observations
- Observation operators are independent of the model and can easily be shared, exchanged, compared

Generic UFO



- Interface classes have to be compatible between models and observation operators
- The UFO is NOT:
  - Attempting to access every possible model grid (or not grid)
  - Attempting to force every model to cast its data structures into a *mother of all data structures* that it can access
- The `State` must be able provide the value of the requested variables at the requested locations

Code Sprint Scope

- The focus of this code sprint will be the observation operator and its interfaces

- Included:
  - Scientific part of observation operators
  - Quality control

- Excluded (but there will be other code sprints):
  - Bias correction
  - Horizontal interpolations
  - IODA (code written will be an example of requirements)

What we have

- Source code repositories:
  - OOPS
  - GSI
  - FV3GFS, WRF, MPAS (if needed)
  - CRTM
  - UFO (empty)

- Data files:
  - Low resolution GSI test case
  - Observation data files (NetCDF and HDF)
  - GFS state values interpolated at observation locations
  - Routines to read/write data files

What we have

- A working environment
  - Repositories to share code
  - Containers to test code
  - Collaborative tools

What we have

- A working environment
  - Repositories to share code
  - Containers to test code
  - Collaborative tools

- And also
  - 12 developers
  - 2 weeks
  - Coffee! (and chocolate)

Preliminary list of tasks

- Learn about
  - OOPS and UFO
  - Development environment
  - Tools for collaboration

- Define data structures for
  - Observations locations
  - Interpolated fields (to obs locations)
  - Observation vectors
  - Simplified observation space

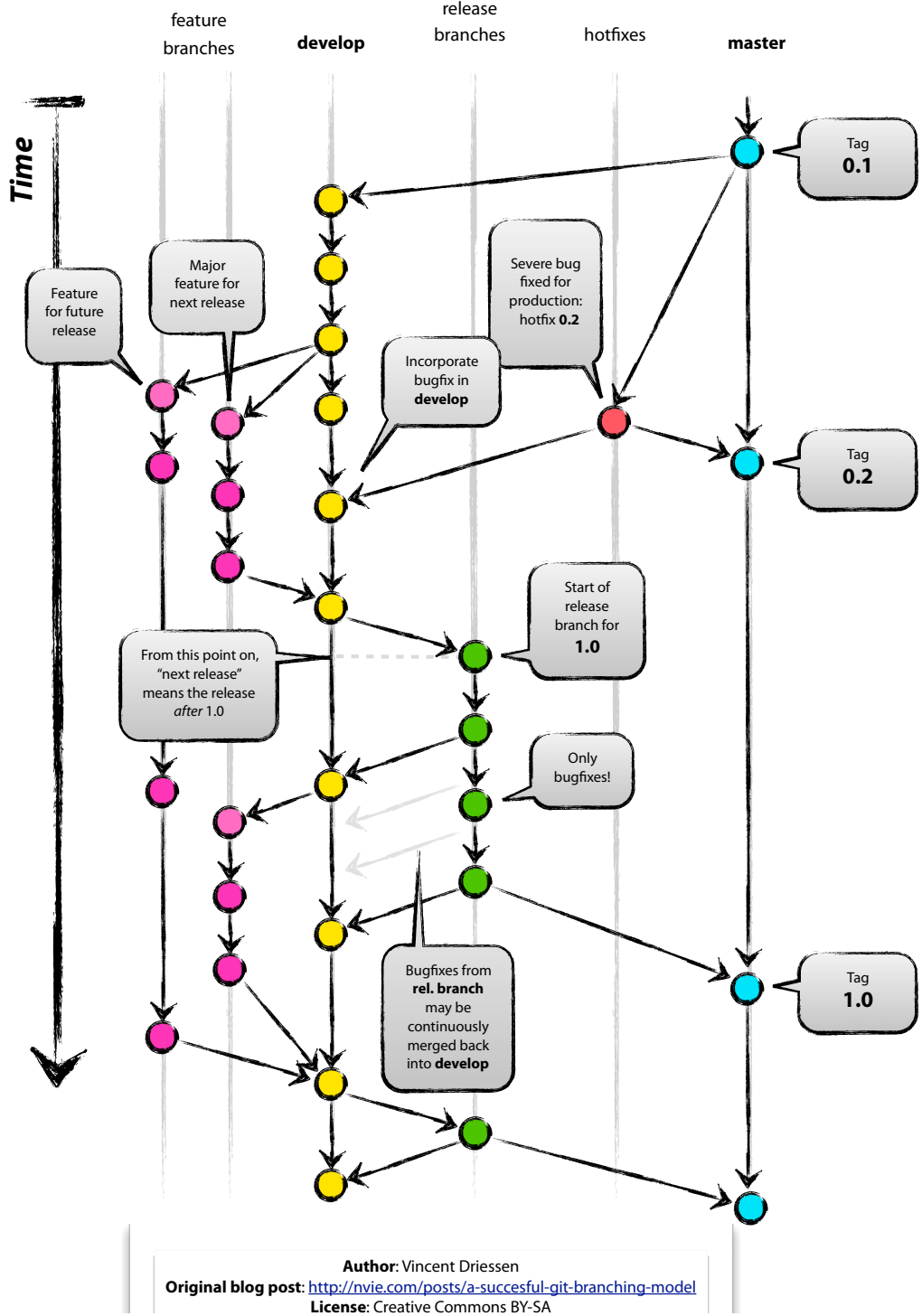
Preliminary list of tasks

- For selected "plain" observation operators:
  - Determine required inputs (fields and metadata)
  - Identify outputs
  - Encapsulate inputs and outputs
  - Interface inputs and outputs with framework
  - Write tests
  - Run tests and validate code

- Add quality control
  - Identify required inputs
  - Evaluate scope for generic QC operators
  - Write tests
  - Run tests and validate code

Tools

Git flow


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0
Author: Vincent Driessen
Original blog post: http://nvie.com/posts/a-succesful-git-branching-model
License: Creative Commons BY-SA

Git flow cheat sheet

Get the code:

```
git clone https://user@github.com/UCAR/oops.git oops
cd oops
```

Get and track the develop branch:

```
git checkout --track origin/develop
```

Start git-flow:

```
git flow init -d
```

Start a new feature branch:

```
git flow feature start mygreatstuff
```

Publish a feature branch to the origin repository:

```
git flow feature publish mygreatstuff
```

Track a feature branch from the repository (after it has been published):

```
git flow feature track otherstuff
```

cmake/ctest

- cmake is a modern open-source system that manages the build process (i.e. generates makefiles) in an operating system and compiler-independent manner.
- OOPS is built with ecbuild, a set of cmake macros.
- It can track dependencies between projects (for example eckit, fckit, oops).
- There is a mechanism for declaring and running unit (and other) tests.

cmake/ctest cheat sheet

Typical first use (from build directory, outside of source):

```
ecbuild -DFCKIT_PATH=${BUILD}/fckit \
        -DECKIT_PATH=${BUILD}/eckit ${SRC}/oops
make -j4
ctest
```

After modifying the code:

```
make -j4
ctest
```

To see full output for one test:

```
ctest -VV -R test_qg_obsoperator
```

The name of the test is a regular expression and behaves as it has a * at the end.

Unit testing

- Using Boost framework
- Easy for plain code
- More complicated with templated classes (99% of OOPS)
- Boost unit test framework is very fussy about compiler versions
- We will invesgigate other options (2018)

Object Oriented Programming

Object Oriented Prediction System (OOPS)

Configuration

- `eckit::Configuration` class
  - Mostly used for holding user defined parameters (21^{st} century `NAMELIST`)
  - Can be constructed from JSON or YAML files
  - No method to modify contents

- `eckit::LocalConfiguration` class
  - Sub-class of `eckit::Configuration`
  - Add methods to modify or add elements
  - As the name implies, the intended use is for local variables (in OOPS arguments are always passed as `eckit::Configuration`)

Configuration examples

Get an element:

```
const std::string bgn(conf.getString("Begin"));
const double tol = conf.getDouble("tolerance");
```

An element can be a subtree:

```
eckit::LocalConfiguration conf;
fullConfig.get("ObsBias", conf);
```

Or vector:

```
std::vector<eckit::LocalConfiguration> obsconf;
conf.get("ObsTypes", obsconf);
```

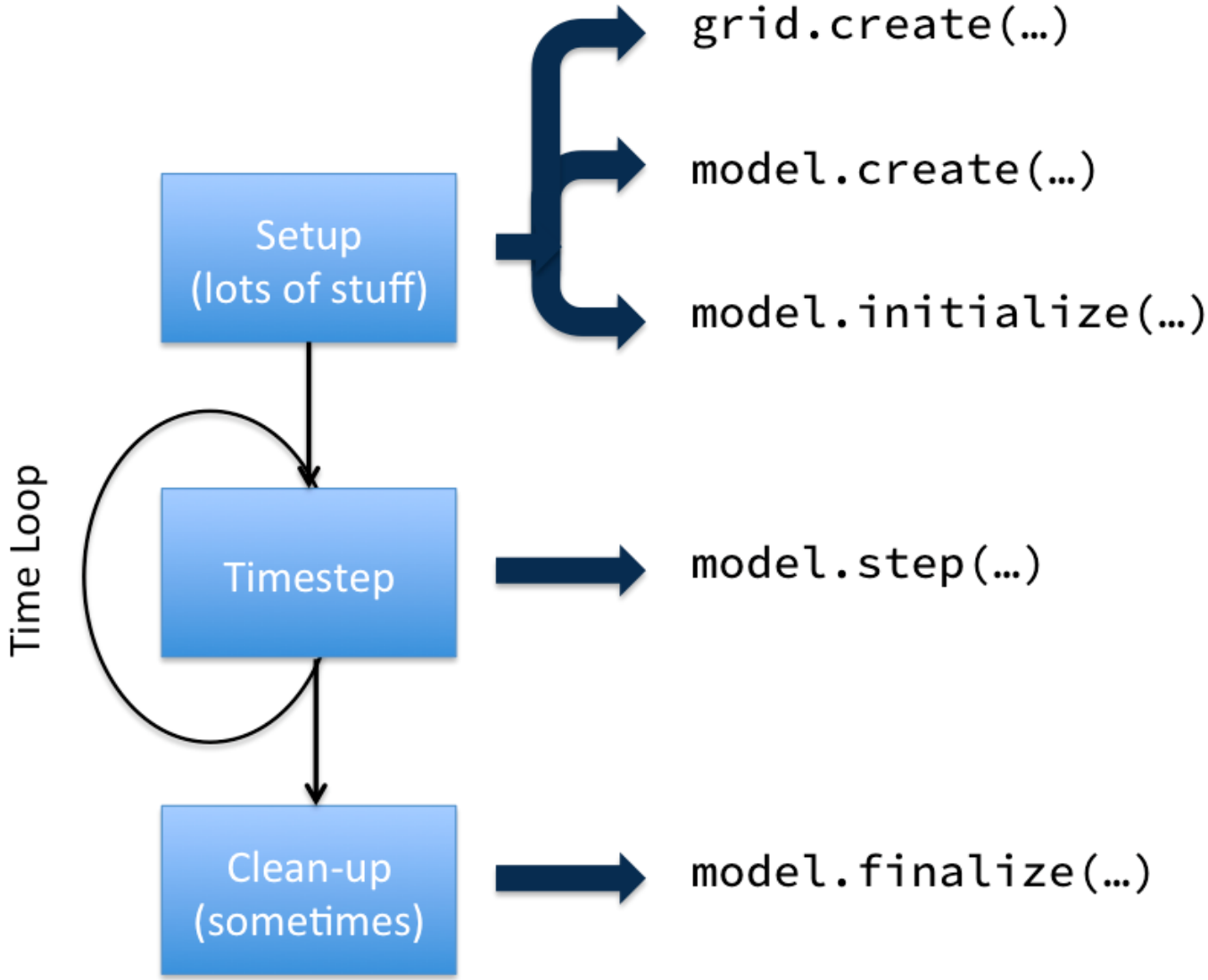
Construct from a part of a larger Configuration:

```
const eckit::LocalConfiguration modelConfig(fullConfig, "Model");
```

Note: get will return an empty `Configuration` if the element does not exist, the constructor will fail.
And there is a Fortran interface!

JSON file

Forecast Model



Forecast Model

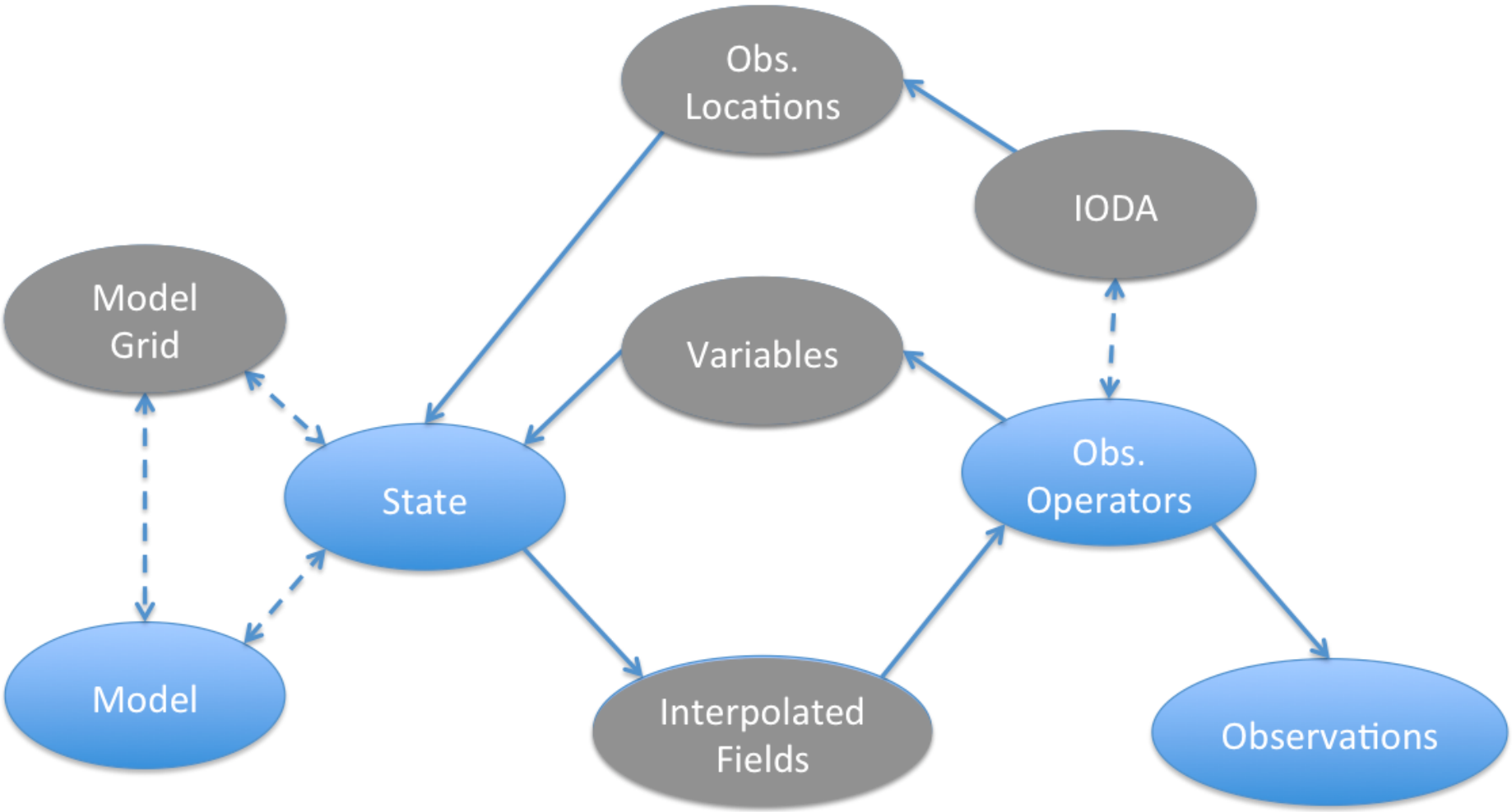
```
Log::info() << "Model: forecast starting: " << xx << std::endl;
this->initialize(xx);
post.initialize(xx, end, model_->timeResolution());
while (xx.validTime() < end) {
  this->step(xx, maux);
  post.process(xx);
}
post.finalize(xx);
this->finalize(xx);
Log::info() << "Model: forecast finished: " << xx << std::endl;
```

- Log is a print for the 21^{st} century: each channel can be directed to stdout, web site, operators, database... without touching the code
- `xx` is the state passed as argument (it is `Printable`)
- The `PostProcessor` is the key to achieve separation of concern. Everything that needs doing during the model integration but is not part of the model (i.e. does not modify the state) is a `PostProcessor`.

H(x): `Observer`, `HofX` and UFO

- The `Observer` class implements the computation of H(x) as the model is running.
- It is a PostProcessor.

H(x) HofX and UFO



- For any realistic application the model is needed.

State

Variables

GeoVaLs

ObservationSpace

ObsVector