OOPS: Object Oriented Prediction System

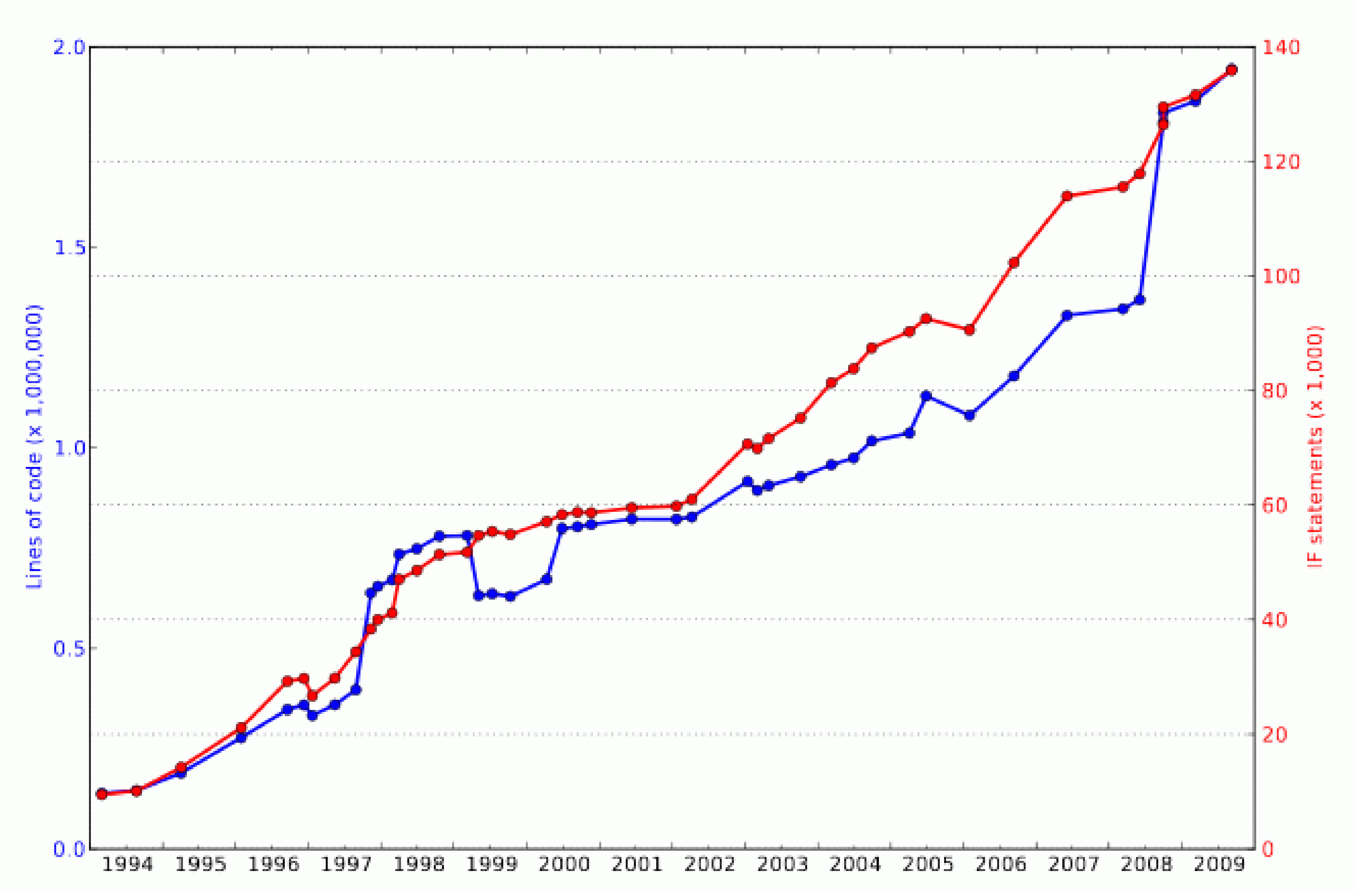
The evolution of the IFS code in the coming 3 years – synthesis of the « OOPS day »

This talk has been prepared by C. Fischer, freely extracting material and summarizing the talks and discussions that have taken place at ECMWF on the November 18th 2009

Why to re-arrange the IFS code ?

- The IFS code has reached a very high level of complexity. However, most configurations and options are set up and defined globally from the highest control level down.
- The maintenance cost has become very high.
- New cycles take longer and longer to create and debug.
- There is a long, steep learning curve for new scientists and visitors.
- It is becoming a barrier to new scientific developments such as long window weak constraints 4D-Var.
- Some algorithmic limitations:
  - Entities are not always independent => H^t R−1 H is one piece (jumble) of code.
  - The nonlinear model M can only be integrated once per execution => algorithms that require several calls to M can only be written at script level.

IFS growth: unfortunately, it's not an investment: It's growth of costs, not of benefits.



Modernizing the IFS

- Re-assess « modularity »:
  - Define self-sufficient entities that can be composed, that define the scope of their variables (avoid « bug-propagation ») => requires a careful understanding and definition of their interface
  - Avoid as much as possible global variables
  - Will require to widen the IFS coding rules and ***break the « setup/module/namelist » triplet paradigm***
- Information hiding and abstraction

The above leads to ***object-oriented programming***

Basics about OO-programming

- One key idea of Object-Oriented programming is to organize the code around the data, not around the algorithms.

- The primary mechanism used by object-oriented languages to define and manipulate objects is the class
- Classes define the properties of objects, including:
  - The structure of their contents,
  - The visibility of these contents from outside the object,
  - The interface between the object and the outside world,
  - What happens when objects are created and destroyed.

More basics about OO

- Encapsulation: content+scope of variables+interfaces (operators) put altogether
- Inheritance: allows more specific classes to be derived from more general ones. It allows sharing of code that is common to the derived classes.
- Polymorphism/Abstraction: ../..

Even more basics about OO

- Polymorphism: refers to the ability to re-use a piece of code with arguments of different types.
- Abstraction: refers to the ability to write code that is independent of the detailed implementation of the objects it manipulates. It allows algorithms to be coded in a manner that is close to their mathematical formulations.

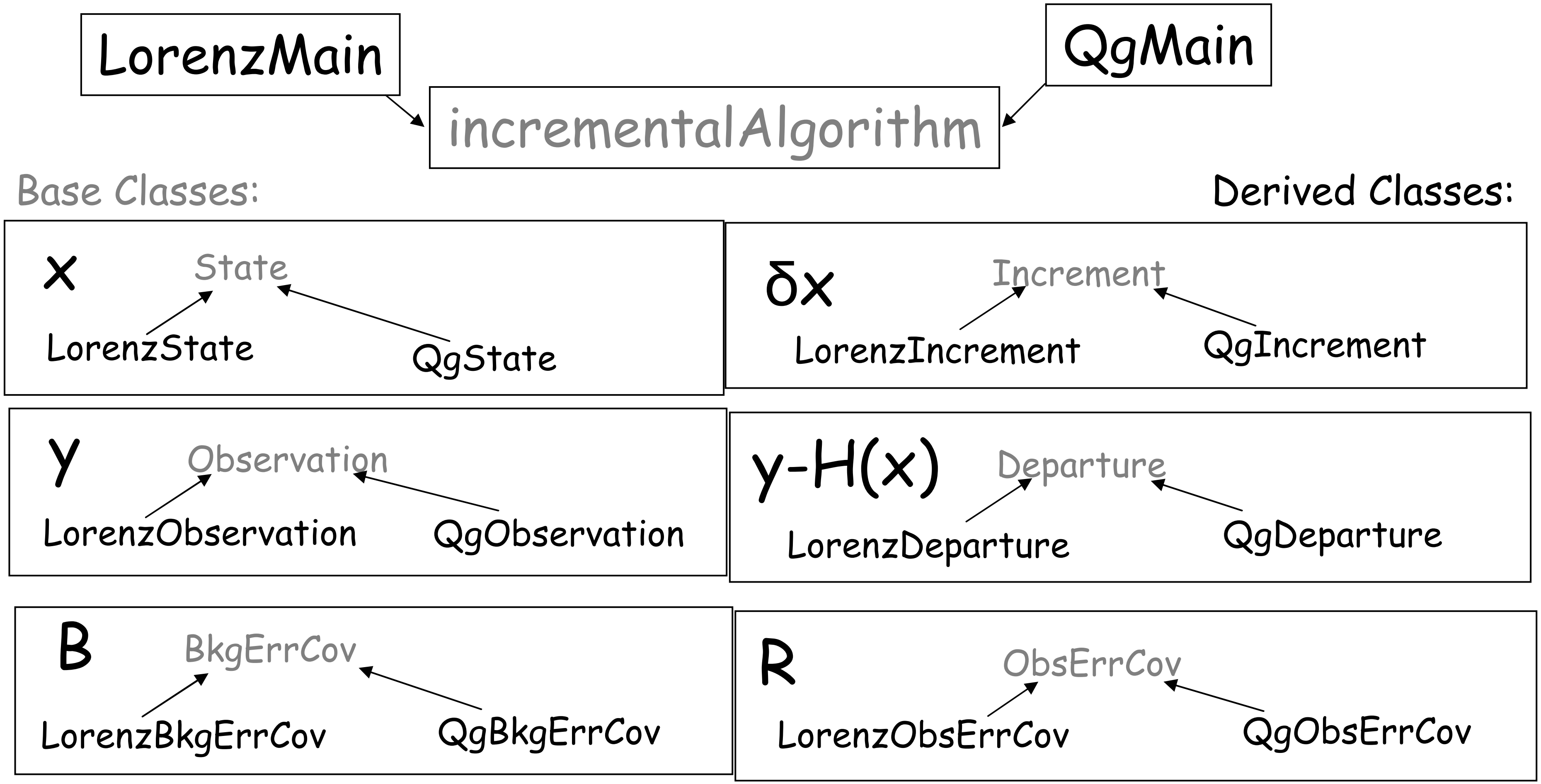
Abstraction: Incremental 4D-Var

```
void incremental_4dvar(CostFunction4dvar & J,
                       ControlVariable & x,
                       Observation & y,
                       int & nouter ) {

  ChangeVariableSqrtCovar chavar(1, *J.B);
  double zj0 , zj1
  int jout;
  int ctlsize = J.B->cvecsize();
  ControlVector dx(ctlsize), gx(ctlsize),
                da(ctlsize);
  dx = 0.0;
  da = 0.0;
  Trajectory traj(J.hmop4d->get_nstep());

  for (jout=0; jout < nouter; jout ++ ) {

    Departure * ydep;
    ydep=J.get_R()->get_dep("ombg");

    Observation * yeqv;
    yeqv=y.clone("obsv");

    // Setup trajectory and departures

    ControlVariable xwork(1,x.get()[0]);
    J.get_hmop4d().nl(xwork,*yeqv,traj);
    ydep->diff(*yeqv,y);
    if (jout == 0) ydep->putdb();
    traj.set(da);
    traj.set(*ydep);
    J.settraj(traj,chavar);

    // compute inital cost and gradient
    dx = 0.0;
    J.simul(dx,gx,zj0);

    // CG Minimization
    CG(J,dx,gx,4);

    // Compute final cost and gradient
    J.simul(dx,gx,zj1);

    // Form increment and analysis
    // in physical space
    Increment * dxtmp;
    dxtmp=J.get_B()->get_inc();
    IncrementalControlVariable xinc(1,*dxtmp);
    chavar.vect2var(dx,xinc);
    *xinc.get()=*xinc.get()+*x.get();
    da = da+dx;
  }

  // Final diagnostics
  ControlVariable xwork(1,x.get()[0]);

  Observation * yeqv;
  yeqv=y.clone("obsv");
  J.get_hmop4d().nl(xwork,*yeqv,traj);
  Departure * ydep;
  ydep=J.get_R()->get_dep("oman");
  ydep->diff(*yeqv,y);
  ydep->putdb();
}
```

Toy OOPS

- 'Toy' data assimilation system to try out Object-Oriented programming for IFS
- Abstract Part
  - Code the algorithm in terms of base classes which serve to define interfaces to the data structures & functions
    - can be compiled separately
- Implementations
  - Code Lorenz and QG models in terms of derived classes from the base classes which define data structures and functions
    - without change of abstract part

Toy OOPS implementations



Testing of Toy OOPS

C++ & Fortran90	Fortran2003
• IBM – xlf90 and xlC	• IBM – xlf – fortran/xlf/12.1.0.4
• NEC – sxf90 and sxc++	• NEC – not available
• Linux – pgf90 and pgcc – gfortran and gcc	• Linux – nagfor – gfortran

Toy OOPS Summary

- Demonstrate writing a data assimilation algorithm in abstract terms such that each part is easily identifiable and switching one part does not mean complete code re-write
- Mixed C++/Fortran90 technically OK
- Compute done in F90 so Gflops same as now
- By design OO layer at top level – for data structure and algorithm definition
- Improve IFS interface to ODB - very suitable for OO

→IFS : a ‘F90 / C++ sandwich’

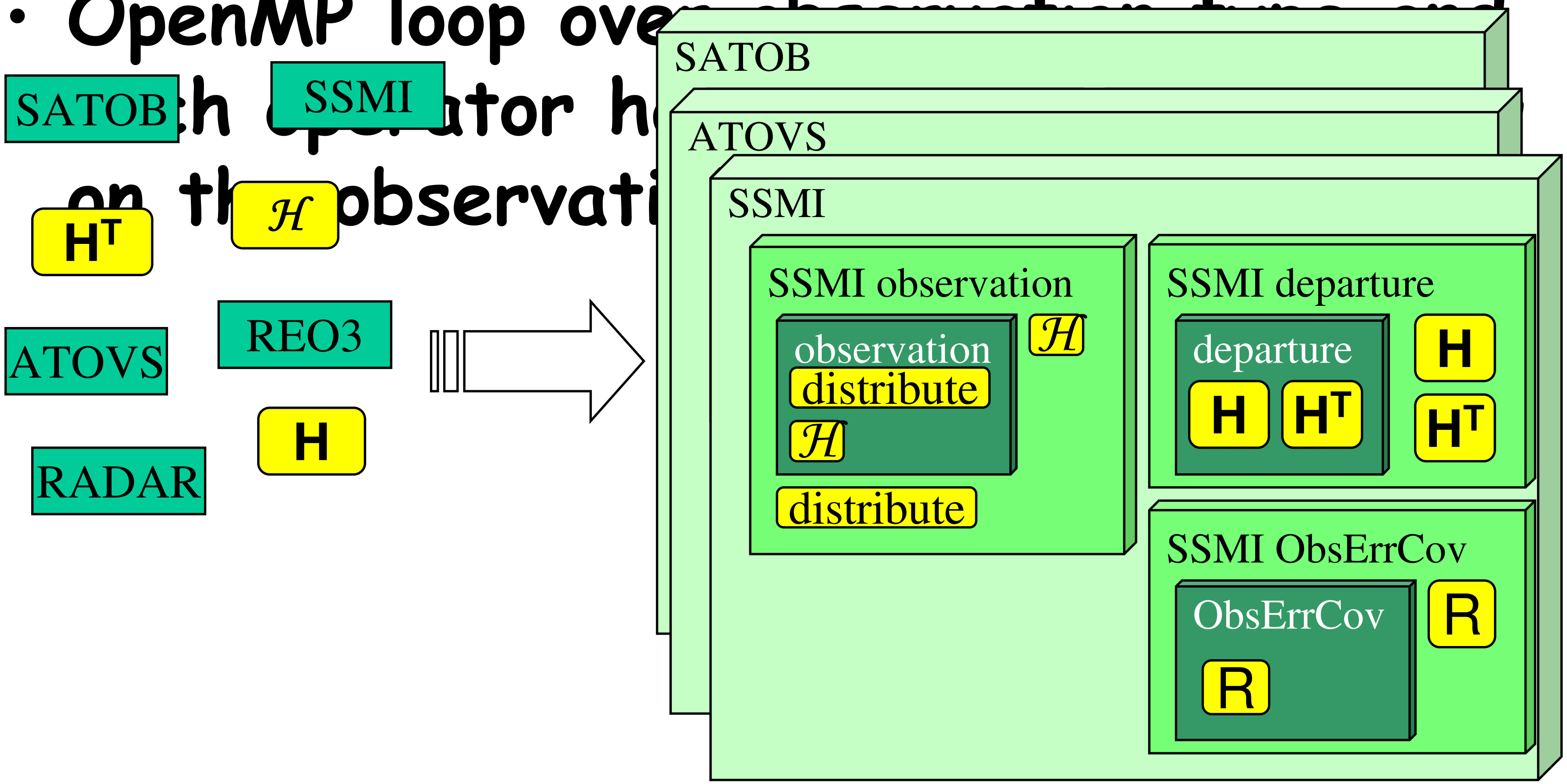
Main program: master.F90
calls mpl_init etc.

Control layer in C++ : IFS_main
Abstract part: IncrementalAlgorithm.cpp,
Stepo.cpp, Hop.cpp,
State.cpp, Increment.cpp, etc.
IFS specific: IFS_State.cpp, IFS_Increment.cpp, etc.

Computational parts in F90:
cpg.F90, callpar.F90, rttov.F90 etc.

Polymorphism

- ODB retrievals in H (hop.F90), H (hoptl.F90), H^T (hopad.F90) depend on the observation type (see ctxinitdb.F90)
- OpenMP loop ove … h … ator h … on th … observati …



What have we learned from the toy system so far ?

- The basic design seems appropriate for our purpose.
- Data assimilation algorithm can be made independent from the model.
- The same basic design can be implemented in Fortran 2003, C++ or a mixture of C++ and Fortran 90. F2003 compilers are still rare (and we are debugging them …). OO programming in C++ requires fewer lines of code than Fortran, but ***Fortran developers will need getting used to its syntax***.
- Tools (debugger, traceback, profilers, MPI, etc...) work for all languages.
- Performance should not be an issue since we only re-code the control level where almost no computing time is spent.

Transition from IFS to OOPS

- The main idea is to ***keep the computational parts of the existing code and reuse them in a re-designed structure*** => this can be achieved by a top-down and bottom-up approach.
- From the top: Develop a new modern, flexible structure => ***Expand the existing toy system***.
- From the bottom: ***Move setup, namelists, data and code together***.
  - Propose new coding guidelines to that effect,
  - Everybody participates by applying it to the part of the code they know.
  - Create self-contained units of code.
- C++/F95 breaking levels: STEPO and COBS/HOP
- Put the two together: Extract self-contained parts of the IFS and plug them into OOPS => this step should be quick enough for versions not to diverge.

Afternoon session: questions and discussions …

- What language ?: combination C++/F95 => some training on C++ coding required, but the first to develop should then teach the others
- User interface:
  - Xml files: incremental rather than full-default; no more namelists after OOPS !!!
  - Must preserve the facility to read in model parameters from a model input file (like with « FA » files; for LAM at least)
  - Interface with Python: possible collaboration with MF's « VORTEX » project
  - Change the S.C.R. tool at ECMWF ?: maybe move to Subversion (already used by Hirlam & M.O. / possibility to have HTML on-line extension)
- Documentation: needs to remain at a reasonable level (clean code is « auto-documentary »)

Afternoon session: follow-on …

- At which level to split OO and standard F ? How far should OO go into the IFS ?:
  - Start with D.A. control; assess the interior of the forecast model(s) later (NL, TL, AD) => timestep organization, externalize physics ?, phys/dyn interface, timestep 1 specificity
  - Break STEPO, make GP buffers the natural vehicle for initializing and passing model data at OO-level (spectral transforms and data become an « optional » entity within the models)
  - Later on, define grids and interpolators as Objects (both « base objects » and « instantiated objects »)
- High-level entities: ocean v/s atmospheric model, EPS and singular vector computation, EnsDA
- For « bottom-to-top » approach: write ***guidelines*** for helping developers to identify their entities

Opportunity v/s risks

- **Opportunity**:
  - Move towards a more “modern” code, sharing more concepts with other system/I.T. codes
  - Guidelines for the bottom-to-top approach will force a general and rather drastic review of the existing code (and options in the code) => some rarely used Research options may disappear !
  - Develop new configurations of the assimilation at the OO-level: NL cost function, hybrid, filters, …
  - Review of the obs operator interfacing, based on a scientific identification of the operators, while totally hiding the ODB database structuring (at the scientific level of the code)
  - Some commonly defined, if not shared, low-level tools of the (otherwise Project-own) user-to-model interface
- **Risks**:
  - Long-lasting efforts that may never end in practice ?
  - Some bets are implicit: future of Fortran programming in Met’ HPC code; actual benefit of OO-concepts once implemented in the whole of the IFS
  - A rather tricky transition period to be organized, but the switch would be “at once” with no backward compatibility (of code) => Research developments will need to be separately adapted
  - Impact on MF and Partner’s applications: especially LAM code

Impact on home/partner applications: a first glance

- **LAM: re-organization of LELAM key**
  - Jb code & control vector handling
  - General strategy for how to arrange LAM specificities in the context of OO (inheritance, polymorphism, … *or some « dirty » tricks to negotiate with ECMWF* ?)
  - Handling of spectral space data in the model & new implementation strategy for biperiodization needed ?
  - « revival » of LRPLANE in the spirit of modular interpolator code
- **MF's own 4D-VAR multi-incremental sequence:** adaptations of Arpège specifities & question of shared C++ assimilation control level
- **adaptation of Full-Pos/e927 with a well-defined interface for OOPS (2-3 possible strategies, to be further decided) =>** ideally, one should be able to almost code the sequence « global forecast + e927 + LAM forecast » within one C++ piece of code
- **Keep the possibility to set up the model parameters by reading from a model input file** (923, (e)927, Arpège and LAM forecasts)
- **DFI code**: Jc-DFI but also regular D.F. initialization in global or LAM models (state vector is both input and output)
- **CANARI**