Implementation of a PDE problem in ObjectMath and Cogito

Deliverable D8.9.4/1

The PREPARE Consortium

Release:	1.6
Date:	April 14, 1996
Status:	draft
Confidentiality:	public
Reference:	PELAB-13-PDE

Contents

List of Figures

1 Introduction

The construction of a program for numerical solution of a system of time-dependent partial differential equations (PDE) is a tedious and error-prone task. Moreover, traditional programming techniques in scientific computing often yield programs that are difficult to *modify*, and to *port* between different (parallel as well as serial) computer platforms.

Recent advances, such as HPF, address the portability issue. However, the problems of easy program construction and modification remain, and call for a raised level of abstraction in programs for scientific computing. Recently, several attempts have been made at introducing object-oriented techniques in this field.

Two efforts along this line are *ObjectMath* and *Cogito*. ObjectMath is a software system for high-level description of mathematical equations. Cogito is an object-oriented software library, implemented in Fortran, for construction of numerical PDE solvers.

This report presents a project in which ObjectMath and Cogito are combined to allow for high-level problem descriptions.

The goal for the project was defined as making it possible to describe a conservative PDE problem in ObjectMath, with subsequent generation of Fortran 90/HPF code. A general numerical solver was created, using Cogito classes, which is linked with the generated code.

1.1 *Summary of results*

To be written

2 Applications

The general solver used in this project can solve conservative problems on multiblock grids. As a typical example, consider the simulation of airflow through an expansion pipe. Figure 1 shows simulated flow in a two-dimensional cross-section of the pipe. There are four plates in the pipe.

The airflow is described by the Navier-Stokes equations. They are valid in the interior of the pipe. At the inlet, outlet, at the plates, and at the pipe walls, boundary conditions are prescribed. Finally, initial conditions describe the airflow at the beginning of the simulation. For more details concerning this application, see [2]

In the present project, we used the same model, but on a simpler pipe which could be described by a one-block grid. Here, there are no inlet and outlet pipes, and no plates. There is one single pipe, with inflow at the left and outflow at the right hand side. The remaining difficulty, from the point of view of code generation, in order to treat the full multiblock application, is the description/specification of boundary conditions.

3 General issues involved

The solver assumes that the PDE problem is on conservative form:

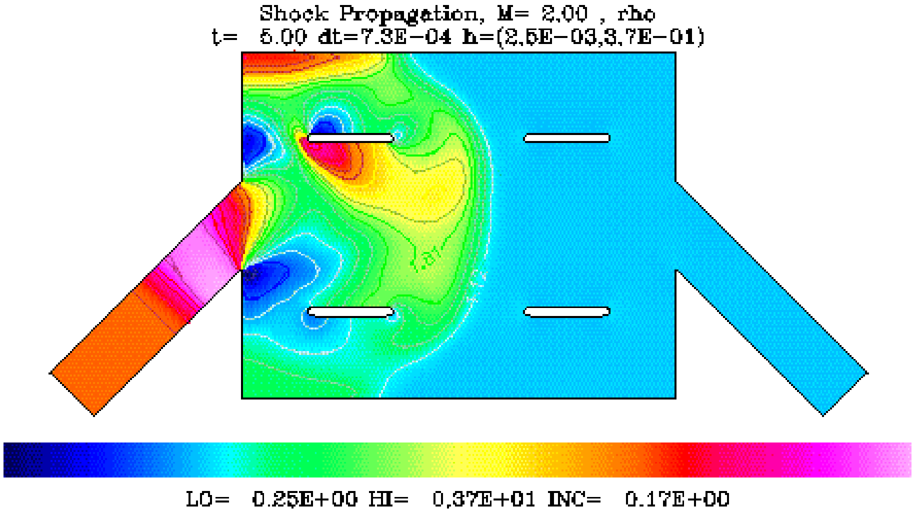

Figure 1: Simulation of airflow through an expansion pipe

$$\frac{\partial Q}{\partial t} + \frac{\partial F(Q)}{\partial x} + \frac{\partial G(Q)}{\partial y} = 0$$

Examples of equations of this kind are the Euler and Navier-Stokes equations in fluid dynamics. F and G are called flow functions. For a 2D problem the unknown state function is defined as

$$Q(t,x,y) = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{pmatrix}$$

where t is the time, x, y space coordinates, $\rho(t,x,y)$ density, $u(t,x,y)$, $v(t,x,y)$ velocities, and $E(t,x,y)$ is the energy.

The difference between the Navier-Stokes and Euler is that Navier-Stokes includes the viscosity (friction) of the fluid into the equation, but Euler doesn't.

The following problem aspects should be described in ObjectMath in order to specify a particular problem of this type.

- The functions F and G.
- Problem specific parameters (for example physical quantities).
- The domain on which the problem is to be solved.
- The boundary conditions.

Ideally, the user should also be allowed to control the details of the numerical method, for example:

- Discretization of space derivatives.
- Time-marching scheme.
- Numerical boundary conditions.
- Step sizes in time and space.

Finally, for full flexibility, the user should also be able to control for example initial conditions and at what points in time values should be stored for post-processing.

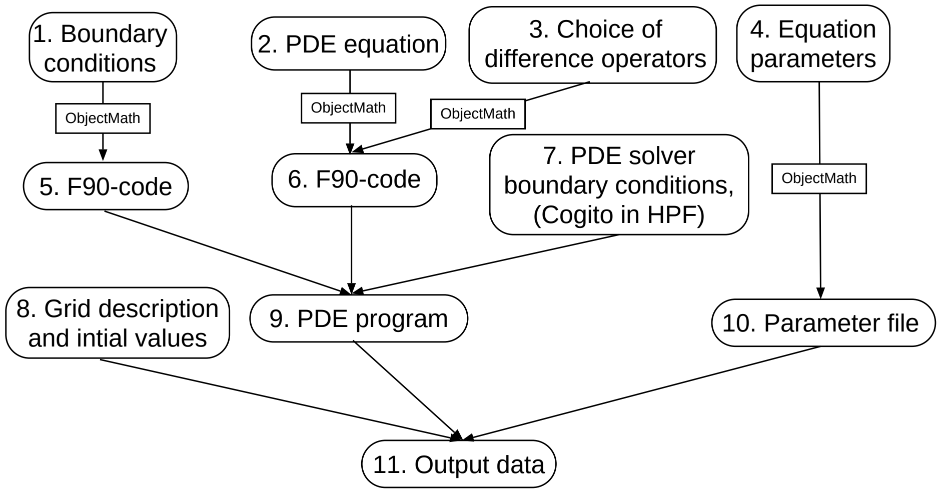

Figure 2: Flow-graph of the definition of a PDE problem

The items in the first row of Figure 2 are specified in ObjectMath. Fortran 90 code is generated from the ObjectMath model and integrated with an PDE-solver, boundary conditions and other data.

In the following section, we discuss a general framework for achieving these goals.

4 Goals of this subproject

An object model that can handle these issues in full generality is presented in [1]. Assuming that the ObjectMath model is specified according to this pattern, a relatively straightforward mapping between the ObjectMath specification and Cogito is possible.

However, generating all problem-dependent code from a full model including constraints, is outside the scope of this subproject. Thus, we restrict the attention to the issue of generating code for the equations, as defined by the functions F and G. The Cogito code only calls code generated form ObjectMath, not vice versa.

In the ObjectMath model, we also include a few other classes, in order to indicate how the general goal can be addressed.

5 Overview of the classes in the PDE ObjectMath model

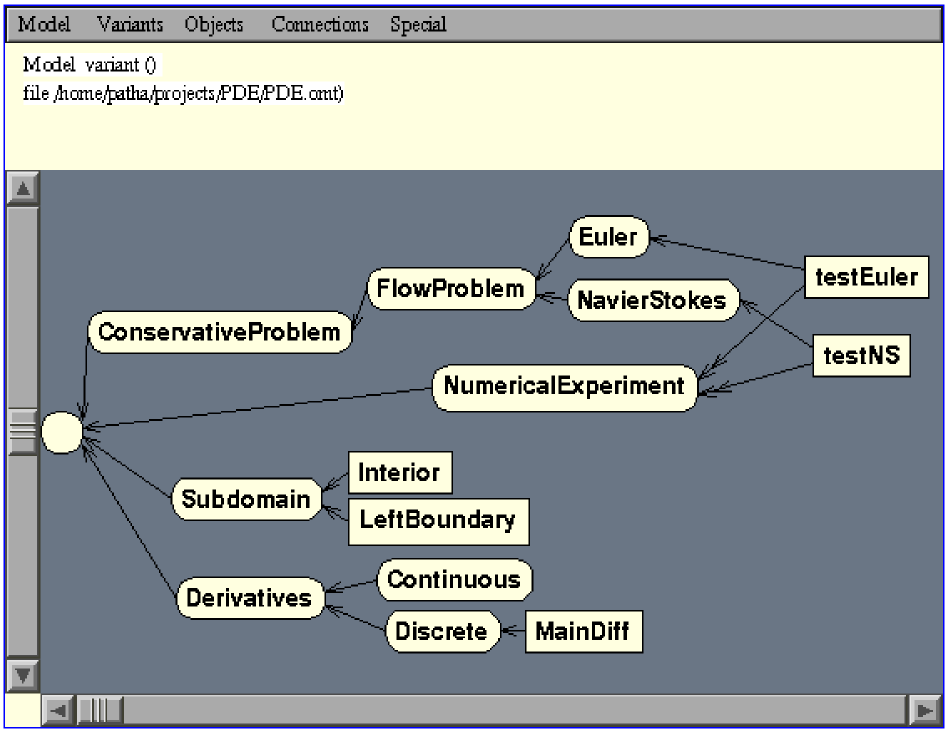

Figure 3: Inheritance structure of the PDE ObjectMath model

Figure 3 shows the inheritance structure of the PDE ObjectMath model. `ConservativeProblem` is an abstract class defining a PDE-equation on the general conservative form shown above. `FlowProblem`, `Euler` and `NavierStokes` define specializations of this equation.

5.1 Specification of the flow functions

`Euler` and `NavierStokes` contains the actual definitions of F and G. F and G are defined in the variables ρ, u, v and E used in the non-conservative formulation. This formulation is then automatically transformed into a formulation where the components of Q (and $\frac{\partial Q}{\partial x}$, $\frac{\partial Q}{\partial y}$ for NavierStokes) are the dependent variables. This is done in ObjectMath using the Mathematica function `Solve`.

A comment concerns problem specific parameters (box 4 in Figure 2). In an application there are normally a number of such values, for example physical quantities such as heat conductivity. Values of these parameters can be specified in ObjectMath code. During code generation, the parameter values can either be used directly in the functions generated (for faster execution) or copied to a file, which is used as input by the solver.

`testEuler` and `testNS` are instances of `Euler` and `NavierStokes`, that can be used for code generation because of their second inheritance from `NumericalExperiment`.

5.2 Grid domains and derivative discretization

The classes `ConservativeProblem`, `NumericalExperiment` and their descendants are sufficient for the present project. However, as mentioned above, a few other classes are included, in order to indicate how the general issues of Section 3 can be addressed.

The class `Derivatives` and its descendants represent derivatives (`Continuous`), or (`Discrete`) difference operators for approximating derivatives. The actual information contained in an object of these classes is a description of what operator to use. This can be mapped to Cogito in a straightforward fashion. The Cogito class *Grid Function* contains a general difference operation, which takes as input the weights to be applied for computing the difference operator at one grid point. Then, the operator is automatically applied at all grid points where it is applicable. The user can also specify that the operator should only be applied in a certain subdomain of the grid.

Objects of the class `Subdomain` represent different parts of the domain. This can be used, e.g., for specifying boundaries of the domain, and, as explained above, for applying different difference operators in different parts of the grid. In particular, regarding the handling of boundary conditions, it is normally necessary to apply special difference operators at the boundaries. Subdomain descriptions in ObjectMath can be mapped to the Cogito class *Grid* which has an operator *Define Subdomain*.

As explained above, Cogito in general handles composite grids. However in this project it is assumed that the domain can be mapped onto a single, rectangular grid. Real word coordinates are used for (x, y). The solver maps the (usually non-rectangular) domain, with a curve-linear grid, to a rectangular area, with a Cartesian grid, see Figure 4.

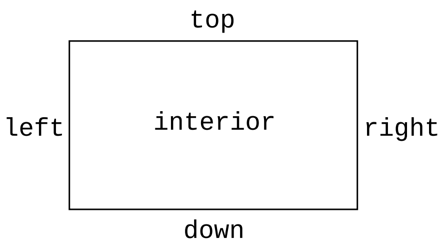

Figure 4: The logical structure of the grid

5.3 Other issues

Boundary conditions, initial conditions, choice of numerical time-marching scheme, etc., can be handled via additional classes, however, these have not been included in the ObjectMath model for this project.

The idea behind our general object model [1] is that the user defines the mathematical problem, and specifies the details of the numerical method by selecting instances of the relevant classes in the model. The class responsible for coordinating the interactions between all the objects involved in a particular case is `NumericalExperiment`.

6 Overview of the classes in Cogito/Grid

Cogito is structured into three layers: one subsystem for computer related classes (Cogito/Parallel), one for grid and grid function classes (Cogito/Grid) and one for classes related to numerical solution methods and PDE problems (Cogito/Solver).

In the present project only the Cogito/Grid tools have been used. Cogito/Grid deals with the management of composite structured grids and grid functions defined on such grids [2]. The task in focus is how to use an *existing* grid. It is assumed that the grid has been generated in advance, by some grid generation package, and is stored on disk.

Cogito/Grid uses an SPMD programming style. Code based on these tools will be portable over a wide range of platforms, serial and parallel.

The Cogito/Grid implementation used in the present project is done in Fortran 90/HPF. The implementation reflects the object-oriented design, making use of the dynamic memory management and data abstraction features of Fortran 90.

The class *Composite Grid* contains a number of blocks. Each block is an object of the class *Grid*. (The blocks can be curvilinear, and are assumed to be logically rectangular.) Moreover, the composite grid contains an object of the class *Interpolation Table*, i.e. a description of how the different blocks relate to each other. The average user will not make explicit references to the interpolation table.

When an object of the class *Grid Function* is created, the corresponding grid is given as a parameter. The grid function will get the same structure (composite or rectangular) as its grid, and—if the grid is distributed—will be distributed in the same way. The user who wants to write a PDE solver at the Cogito/Grid level of abstraction will mainly operate on grid functions. Currently, only explicit difference methods are supported. (Implicit methods fit into the same framework, but require additional classes and operations.) There are operations for inspecting properties of the grid function, and for performing arithmetic operations. In particular, there are methods for applying difference operators to the grid function.

It is possible to define sub-domains of a grid. The arithmetic operations on the class *Grid Function* take a subdomain identity as input. An operation is performed only on the specified subdomain. By this mechanism, boundary conditions can be conveniently implemented.

7 Subtasks that have been addressed

For the generation of code for F and G, the following approach was taken: From the ObjectMath descriptions, Fortran 90 functions are generated, one for F and one for G. These Fortran functions evaluate F and G, respectively, for a single grid point. The Cogito class *Grid Function* has an operator *Apply Function*, which takes as input a function defined for a single grid point, and applies it to all points in the grid. Through a call to this operator, the solver computes F and G on the entire grid. We use an HPF based version of Cogito. Thus, the generated PDE solver will be portable over serial and parallel platforms.

In order to achieve this, the following tasks have been carried out, as parts of this project:

- *Adapting Cogito/Grid to Fortran 90/HPF.* At the beginning of the project, the complete Cogito/Grid layer existed in a Fortran 77 implementation, based on Cogito/Parallel. We made a new implementation of Cogito/Grid, in Fortran 90, removing the connection to Cogito/Parallel. In the new implementation, HPF directives are used for parallelization.

- *Creating a solver framework for conservative PDE problems.* In the Cogito project, a 2D compressible Navier-Stokes solver, on a multi-block grid, has been used as a test bed. In this project, a modified version of this solver was created, where all Navier-Stokes specific elements where isolated and removed, so that a solver framework for conservative problems was obtained.

8 Results

8.1 Examples of generated code

For each function generated, ObjectMath outputs a Fortran subroutine.

The following example is the F function from the Euler case in the ObjectMath model.

```
subroutine testEuler_TF(om_01, P_D_TQ)
use om_PDE, om_this => testEuler_TF
  implicit none
  real(om_real), intent(out) :: om_01(4)
  real(om_real), intent(in) :: P_D_TQ(4)
  real(om_real) om_T1
  real(om_real) om_T2
  real(om_real) om_T3
  real(om_real) om_T4
  real(om_real) om_T5
  real(om_real) om_T6
  real(om_real) om_T7
  real(om_real) om_T8
  real(om_real) om_T9
  real(om_real) om_T10
  real(om_real) om_T11
```

```
  real(om_real) om_T12
  real(om_real) om_T13
  real(om_real) om_T14
  real(om_real) om_T15
  real(om_real) om_T16
  real(om_real) om_T17(4)

  om_T1 = P_D_TQ(2)
  om_T2 = testEuler_TM0**(-2)
  om_T3 = P_D_TQ(1)
  om_T4 = om_T3**testEuler_Tgamma
  om_T5 = om_T2*om_T4
  om_T6 = om_T5/testEuler_Tgamma
  om_T7 = om_T1**2
  om_T8 = om_T7/om_T3
  om_T9 = om_T6+om_T8
  om_T10 = P_D_TQ(3)
  om_T11 = om_T1*om_T10
  om_T12 = om_T11/om_T3
  om_T13 = P_D_TQ(4)
  om_T14 = om_T6+om_T13
  om_T15 = om_T1*om_T14
  om_T16 = om_T15/om_T3
  om_T17 = (/ om_T1, om_T9, om_T12, om_T16 /)
  om_O1 = om_T17
end subroutine  testEuler_TF
```

The first parameter `om_O1` is the result vector and `P_D_TQ` is the in parameter Q. The mathematical expression in the function body is processed by a common subexpression eliminator and all intermediate results are stored in temporary variables.

Values to the two physical parameters `testEuler_TM0` and `testEuler_Tgamma` can be given in the Fortran code or assigned values before code generation in ObjectMath.

8.2 Examples of Cogito HPF code

8.3 *Performance comparison between generated and handwritten code*

Here we compare an old handwritten version of the PDE equation to a version generated by ObjectMath (only serial execution). The test was done on a Digital AlphaServer 2000 5/250 with the DEC Fortran 90 compiler. The grid was composed of 201x201 grid-points and the solver was stopped after five time-steps. The time to write the result to disk is excluded. The timing is performed five times in sequence to reveal random variations. All numbers are in seconds.

The numbers at the bottom is the mean value. The generated version shows out to be 7% slower than the handwritten.

Handwritten	Generated
50.48	50.43
49.70	53.19
47.59	54.04
49.75	53.06
49.79	53.17
49.46	52.78

Table 1: Time in seconds for handwritten compared to generated code

8.4 *Performance comparison between HPF and message passing code*

To be written

8.5 *Performance scalability*

To be written.

References

[1] M. Thuné and K. Åhlander *Towards an Expressive Language for PDE Solvers* Accepted for presentation at the European Symposium on Programming (ESOP'96), Linköping University, April 22 - 26, 1996

[2] J. Rantakokko, *Object-oriented software tools for composite-grid methods on parallel computers.* Report 165, Dept. of Scientific Computing, Uppsala University, Uppsala, 1995.

A The ObjectMath model

```
model PDE;
  packages "CodeGeneration`";

  (* INSTRUCTIONS !! *)
  (* Perform code generation with
     testEuler`run or testNS`run *)

class ConservativeProblem (* Coordinates: -12, 128 *)
  expose:
  dt;
  dx;
  dy;
  PDEequation := dt[Q] + dx[F] + dy[G] == 0;
```

```
end;

class FlowProblem inherits ConservativeProblem
(* Coordinates: -7, 25 *)
  expose:

  symbol rho, u, v, Energy;

  (* kernel expressions *)
  Fstd := { rho*u, rho*u^2+p, rho*u*v, u*(rho*Energy+p) } ;

  Gstd := { rho*v, rho*u*v, rho*v^2+p, v*(rho*Energy+p) } ;

  (* methods for equation of state *)
  p;
  setEquationOfState[a_] := (p := a);

  (* conversion form the nonconservative variables above
     to the conservative representation Q *)
  q1 := Array[qc,4];
  q2 := {rho, rho*u, rho*v, Energy*rho};
  vars := {rho,u,v,Energy};

  invq := Solve[ q1==q2, vars][[1]] ;
end;

class Euler inherits FlowProblem (* Coordinates: 4, 43 *)
  expose:
  { real array[4] } F[ real array[4] Q_] :=
    Fstd
    /. invq
    /. MapThread[Rule, {q1, Array[Q[[#]]&,4]}]
  ;

  { real array[4] } G[ real array[4] Q_] :=
    Gstd
    /. invq
    /. MapThread[Rule, {q1, Array[Q[[#]]&,4]}]
  ;

  p := 1/(gamma*M0^2)*rho^gamma;

  (* parameters read from file *)
  real gamma;
```

```
  real M0;
  parameters := { gamma, M0 };
end;

instance testEuler inherits Euler, NumericalExperiment
(* Coordinates: 75, -23 *)
  expose:
  run := (

    (* print the functions to generate *)
    Print[F[Array[q,4]]];
    Print[G[Array[q,4]]];

    (* generate Fortran code for F and G *)
    gen;
  );
end;

class NavierStokes inherits FlowProblem
(* Coordinates: 3, -20 *)
  expose:

  { real array[4] } F[ real array[12] Qd_ ] :=
    Fstd - (1/R)*Fvisc
    /. invdqdx /. invdqdy /. invq
    /. MapThread[Rule, { Join[q1,dqdx1,dqdy1],
         Array[Qd[[#]]&,12] }]
    // Simplify
  ;

  { real array[4] } G[ real array[12] Qd_] :=
    Gstd - (1/R)*Gvisc
    /. invdqdx /. invdqdy /. invq
    /. MapThread[Rule, { Join[q1,dqdx1,dqdy1],
        Array[Qd[[#]]&,12] }]
    // Simplify
  ;

  Fvisc := { 0, tauXx, tauXy, f4 };

  Gvisc := { 0, tauYx, tauYy, g4 };

  tauXx := 2*mu*Dt[u,x] + lambda*(Dt[u,x] + Dt[v,y]);
```

```
tauXy := mu*(Dt[u,y] + Dt[v,x]);

tauYx := tauXy;

tauYy := 2*mu*Dt[v,y] + lambda*(Dt[u,x] + Dt[v,y]);

lambda := -2/3*mu;

f4 := u*tauXx + v*tauXy + (k/(Pr*(gamma-1)))*
      Dt[a2,x, Constants->parameters];

g4 := u*tauYx + v*tauYy + (k/(Pr*(gamma-1)))*
      Dt[a2,y, Constants->parameters];

a2 := gamma*(p/rho);

p := Solve[ rho*Energy==unk/(gamma-1)+rho/2*(u^2+v^2), unk ]
     [[1,1,2]];

(* parameters read from file *)
real k;      (* heat conductivity *)
real mu;     (* viscosity coefficient *)
real gamma;  (* ratio between heat capacities at
                constant pressure and volume *)
real R;      (* Reynolds number *)
real Pr;     (* Prantl number *)
parameters := { k, mu, gamma, R, Pr };

(* code optimization by giving the parameters their
   values before code generation *)
parameters := {};
k := 0.0;
mu := 1.0;
gamma := 1.4;
R := 9600.0;
Pr := 0.72;

(* conversion from derivatives of "vars"
   to derivatives of Q *)
dqdx1 := Array[dqdxc,4];
dqdx2 := Dt[q2,x];
invdqdx := Solve[ dqdx1==dqdx2, Dt[vars,x] ][[1]]
  /. MapThread[Rule, { Dt[vars,x], Dt[vars,x,___] }];
  (* cludge for matching the Constants -> { ... } option *)
```

```
  dqdy1 := Array[dqdyc,4];
  dqdy2 := Dt[q2,y];
  invdqdy := Solve[ dqdy1==dqdy2, Dt[vars,y] ][[1]]
    /. MapThread[Rule, { Dt[vars,y], Dt[vars,y,___] }];
    (* cludge for matching the Constants -> { ... } option *)
end;

instance testNS inherits NavierStokes, NumericalExperiment
(* Coordinates: 28, -32 *)
  expose:
  run := (

    Print[F[Array[q,12]]];
    Print[G[Array[q,12]]];

    gen;
  );
end;

class Derivatives (* Coordinates: 39, -79 *)
end;

class NumericalExperiment (* Coordinates: 186, 0 *)
  expose:

  gen := (
    (* supress warnings from code generation *)
    Off[Part::partd];

    CodeGeneration`OpenFile["flowFcn","Fortran90"];

    (* generate declartion for all parameter variables *)
    If[ Length[parameters]=!=0,
      CodeGeneration`GenerateVariable[this]];

    CodeGeneration`GenerateEvaluateFunction[F];
    CodeGeneration`GenerateEvaluateFunction[G];

    CodeGeneration`CloseFile[];

  );
end;

class Continuous inherits Derivatives
```

```
(* Coordinates: 15, 24 *)
  expose:
end;

class Discrete inherits Derivatives
(* Coordinates: 20, -32 *)
  expose:
end;

class Subdomain(domainNumber)
(* Coordinates: 36, -107 *)
end;

instance LeftBoundary inherits Subdomain(1)
end;

instance Interior inherits Subdomain(0)
end;

instance MainDiff inherits Discrete
end;
```