Abstract

In this report we provide an introduction to ODE, then present an extensible Object-Oriented framework – written in C++ – with emphasis on the reusability of modules for ODE solvers. The ability to extend this API to accommodate new algorithms as they are developed is particularly attractive. This facilitates our work to find the best numerical method and speed the development of a dedicated simulator for specific cases.

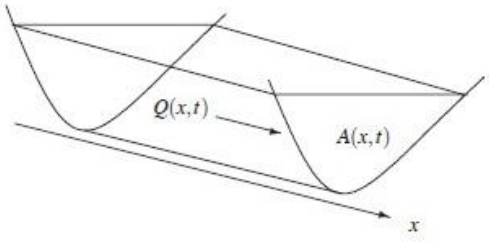
Mathematical Model

Conservative Form of St-Venant Equations

From the fluid mechanics point of view flood propagation is an extremely complicated phenomenon involving the dynamics of a fluid with a free boundary in intense turbulent motion under the acceleration of gravity. When trying to describe mathematically this situation, one is faced with solving a full three-dimensional unsteady Navier-Stokes problem with a free boundary. Under certain assumptions (St-Venant approximation), one may resort to a simpler mathematical representation of the physical reality. This is certainly the case in many engineering applications.

We therefore obtain a set of coupled equations which are a balance between the forces of gravity, friction and inertia applied to the mass of moving water. These equations describe balances between the different processes: diffusion, convection (transport). Depending on whether one of these processes dominates over the others this changes the physical nature of the equation. For example, and this is the case that interests us, when convection dominates, we are dealing with a propagation phenomenon.

The one-dimensional open flow equations are described in terms of water depth and flow rate, and the evolution of these quantities satisfy the St-Venant equations, which simply express the conservation of mass and quantity of movement in the direction of flow (called a system of conservation laws).



The S-Venant system can be written in different forms, we consider the conservative form of the equations.

$$\partial_t \boldsymbol{U} + \partial_x \boldsymbol{F}(\boldsymbol{U}) = \boldsymbol{S} \qquad (1)$$

where

$$\mathrm{U} = \begin{pmatrix} \mathrm{A} \\ \mathrm{Q} \end{pmatrix}$$

$$F = (Q, Q^2/A + gI_1)$$

$$S = \left(0,\ gI_2 + gA\left(S_f - S_b\right)\right)$$

where F and S are flux tensor and source terms respectively.

- **U**: state vector of the state variables, respectively wetted area and discharge of section flow
- **F**: physical flux, convective and pressure
- Sf: friction term expressed by the Manning formula
- Sb : bottom slope term expressed as derivative of the bathymetry
- **A**: wetted cross-section area which depends $\left(x, h(x,t)\right)$
- **g**: acceleration of gravity

I_1: pressure term given by $\int_0^{h(x)} (h(x) - \vartheta)\ \sigma(x,\vartheta)\, d\vartheta$

I_2**:** and given by $\int_0^{h(x)} (h(x) - \vartheta)\ \left[\left(\frac{\partial \sigma(x,\vartheta)}{\partial x}\right)\right] d\vartheta$

with 'h' water depth, σ width for a fixed depth and ϑ depth integration variable along y axis.

The finite volume method can be considered as a finite difference method applied to the differential conservative form of conservation laws written in an arbitrary coordinate system. These finite volume schemes approximate the integral form of the conservation laws. At each time step, we solve the integral average (the dependent variables are approximated by integral averages, evolution of the integral average) of the flow variables in each volume.

There are several ways to derive and formulate the equations of fluid mechanics. They can be deduced from the fact that a physical system's behavior is completely determined by conservation laws. Specifically, several properties such as mass, momentum, and energy are conserved. These conditions do not completely determine the behavior of a physical system. It is necessary to add a state law and, for the problem to be well posed, initial conditions and boundary conditions.

The conservation principle gives us an expression for the variation of a quantity within a volume, including the effect of external forces (to be rephrased). The writing of the conservation law in its general form for a quantity U, scalar or vector quantity per unit of volume, acting on a volume Ω in space and bounded by the surface S is as follow

$$\frac{\partial}{\partial t} \int_{vol} U\, d\Omega + \oint_{surf} \vec{F} \cdot d\vec{S} = \int_{vol} Qv\, d\Omega + \oint_{surf} Qs\, d\vec{S} \qquad (1)$$

where U is a vectorial quantity, F and Qs are tensors and Qv is a vector. Technique by which the integral formulation of conservation laws is discretized directly in physical space. The finite volume method directly uses the conservative form of the equation

$$\frac{\partial}{\partial t}\int_{vol} U\, d\Omega + \oint_{surf} \vec{F}\cdot d\vec{S} = \int_{vol} Q\, d\Omega \qquad (2)$$

By Gauss theorem, we have:

$$\frac{\partial}{\partial t}\int_{vol} U\, d\Omega + \oint_{surf} \vec{\nabla}\cdot \vec{F}\, d\Omega = \int_{vol} Q\, d\Omega \qquad (3)$$

differential form

$$\frac{\partial U}{\partial t} + \vec{\nabla}\cdot\vec{F} = Q \qquad (4)$$

where U is a scalar field.

The computational domain is partitioned into a finite number of control volume $[x_{i+1/2}, x_{i-1/2}]$ around a nodal position x_i. The position of the right face $x_{i+1/2}$ is defined as $(x_i + x_{i+1})/2$. The differential system is integrated over each control volume to produce the discrete equivalent of the conservation

$$U_i^{n+1} = U_i^n + \frac{dt}{dx}\left[\left(F_{i-1/2} - F_{i+1/2}\right) + Q_i^n\right]$$

with Q is the source term. This integration technique forms the basis of what is known as the finite volume method. The specific difference between various finite volume schemes is the way in which they approximate the interface convective flux

$$F_{i+1/2} = F\left(U\left(x_{i+1/2}, t\right)\right).$$

where U is the state variables and F the physical flux of the state's variables. This equation expresses simply that the quantity U inside a volume depend only on the flux at the surface (no source inside the volume). We now have an equation which expresses the time evolution of a mean cell value in terms of the flux.

Simulation System (Framework) SFX Package

We present an OO analysis and design of an Object-Oriented framework in physics modeling. This framework implemented is essentially a collection of C++ classes organized in libraries, contains basic building blocks for the numerical solution, corresponding to the explicit finite difference scheme. Solve of the one-dimensional shallow-water equations on the Dam-Break Problem by an explicit numerical method. We have put special effort into making the framework flexible and extendible, which means that it is possible to program new algorithms with little coding by using already tested components. We stress the ability to build a new physical algorithm quickly by using already developed and tested components.

In industrial research projects, the physicist is often called upon to test or experiment with different scenarios; this type of environment meets this need.

Requirements Definition

- **It shall be possible to change the numerical method we use**

We may have several different reasons for wanting to change the numerical method. Maybe, the numerical method yields wrong answers for the particular problem. This includes unstable solutions, solutions that are mathematically correct but physically incorrect, or solutions that are not accurate enough. Another reason for changing the method is efficiency.

- **It shall be easy to construct a program that solves the Dam-Break Problem**

A user shall be able to describe new problems, or new solution methods, and add these to the system. This is particularly useful if, for instance, the user develops a new numerical method and wants to compare this method against other methods.

- **The system shall be flexible. Components shall be changeable**

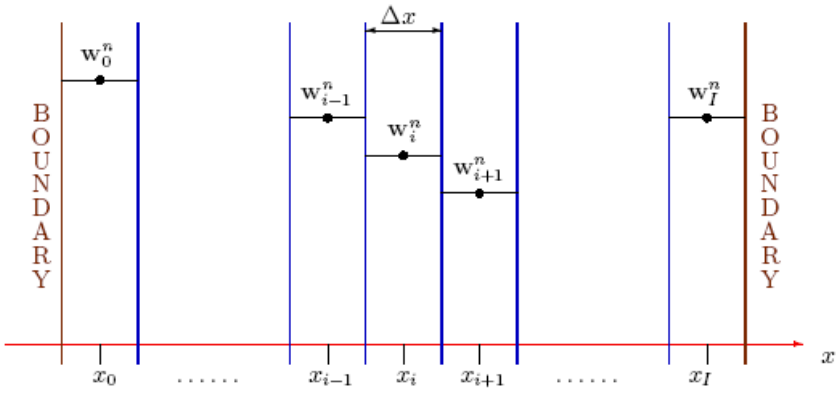
All pieces of the physical algorithm (solve numerically the PDE) shall easily exchangeable. Particularly, it is easy to switch between different implementations of the explicit integrator, or to switch between a flux algorithm (or numerical scheme). When validating numerical schemes, it would be nice to mix different alternatives and compare.

- **It shall be easy to extend the system with new components**

The flux algorithm could be implemented with a Godunov-type scheme or with an ENO flux extrapolation. We need a set a high-level abstraction which identify (represent) the key concepts of the application and let the specific (detail) to subclass;

Key Concepts of Our Application

Main concept of discretization (transformation from continuum to discrete space).



- **Global discretization concept can be characterized by three features:**

The domain of the problem is represented by a collection of simple domains, called cells. In the 1-D case the domain [a,b] of the equation is discretized as a series of points x_i, i=0,...,N-1 where the solution "w" of the equation is discretized as;

Over the cell, the physical process is approximated by functions of desired type (polynomials or otherwise), and an algebraic equation relating physical quantities at selective points, called nodes, of the element are developed; The discretized equations is constructed over each cell;

- **Space discretization concept**

Space discretization consists of setting up a mesh or a grid by which the continuum of space is replaced by a finite number of points where the numerical values of the variables will have to be determined

- **Time Discretization concept**

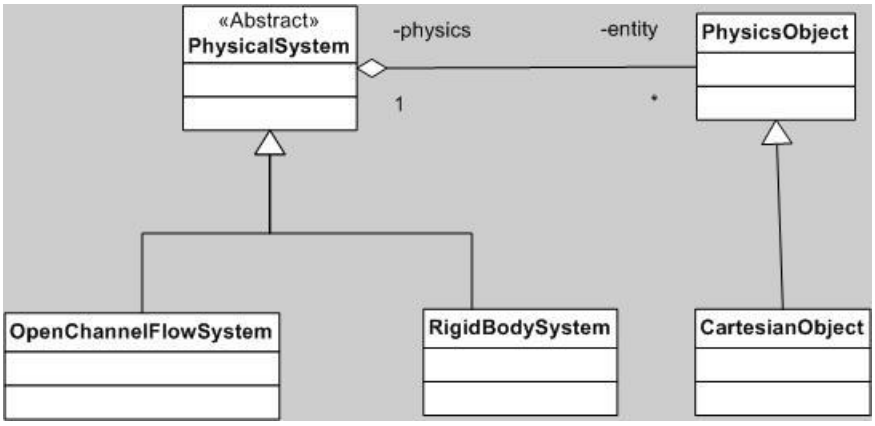
The time discretization is done by explicit time stepping. In the case of explicit scheme unknown variables at the current time level depends on the previous time level.

- **Discrete representation concept**

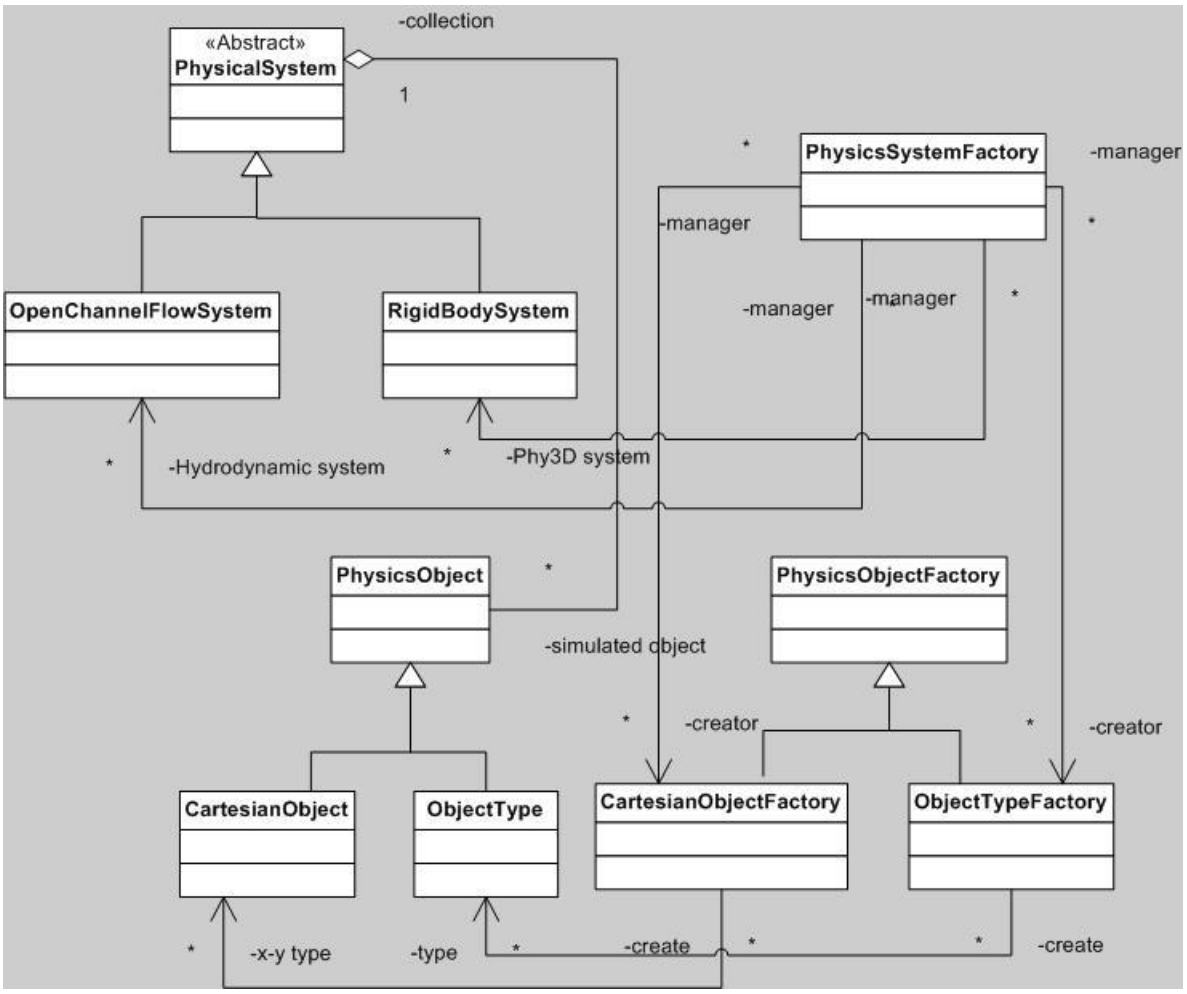
Once the mesh has been defined the equations can be discretized, leading to the transformation of the differential or integral equations to discrete algebraic operations involving the values of the unknowns at the mesh points. The basis of all numerical methods consists of this transformation of the physical equations into an algebraic, linear or non-linear, systems of equations.

Physical System

Every problem has some type of physical object that is in the system of study.

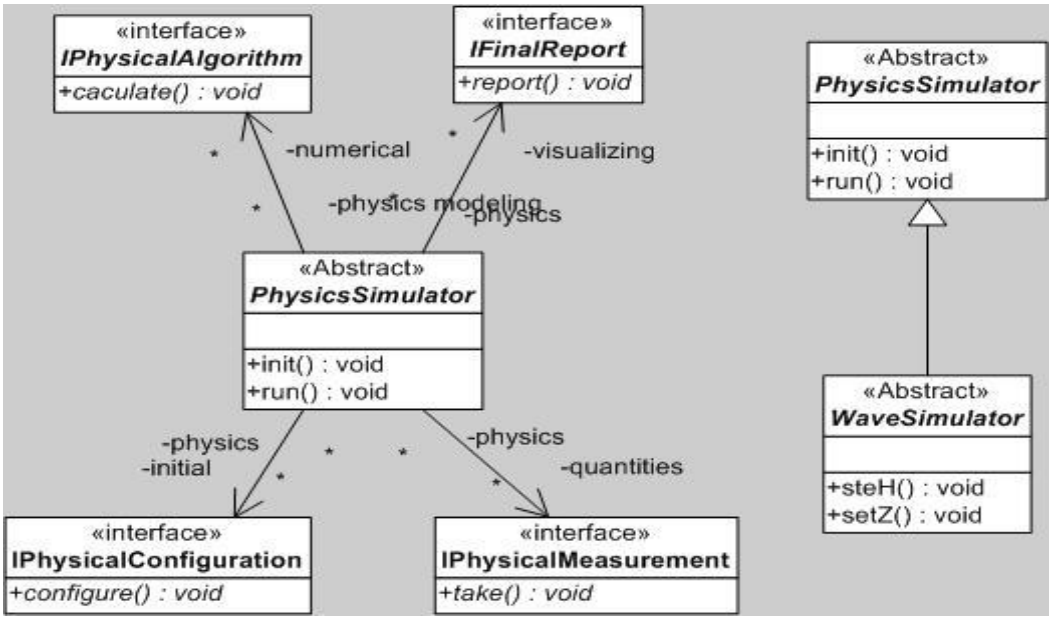


The **PhysicalSystem** and **PhysicalObject** are the top-level classes in our design. The relationship between a **PhysicalSystem** and a **PhysicalObject** is a has an aggregation relationship as indicated by the diamond. This means that PhysicalSystem is made up of or composed of **PhysicalObjects**. How does the Framework use these two classes and expand on them as the basis? A factory is designed to help create those objects. Likewise, for **PhysicalObjects**, an abstract factory is used to create them. A more detailed UML class diagram is shown below:



These are some of the classes that make up the SFX package. The top level to the diagram is the **PhysicalSystemFactory**. This class is a concrete class and is responsible for creating the user specified **PhysicalSystem**. It does so by creating an instance of the user specified PhysicalSystem and adds to it the specified number of **PhysicalObjects** using the appropriate **PhysicalObjectFactory**.

The next part to understand is the relationship between the remaining classes in the system package and their relationship to the **PhysicsSimulator** in the SFX package. The UML diagram below shows them:



The classes on the right side of the arrows are Interfaces. An interface specifies a public contract that the implementing class of it must agree to. Physics Simulator class has two responsibilities. First it creates instances of the user specified classes that implement the interfaces shown in the figure above. It is also responsible for invoking methods to perform the actions of each class that implements the interface at the appropriate time.

The four interfaces shown here are the ones that each developer intending to use the Framework will have to provide implementations for. The following table provides more information about each interface.

Interface	Method	Arguments
IPhysicalConfiguration	configure()	PhysicalSystem ps
IPhysicalAlgorithm	calculate()	Simulation sim PhysicalSystem ps
IPhysicalMeasurement	Take()	Simulation sim, PhysicalSystem ps,
IDDataStore	toString(), toSql()	None,None
IFinalReport	report()	Simulation sim PhysicalSystem ps

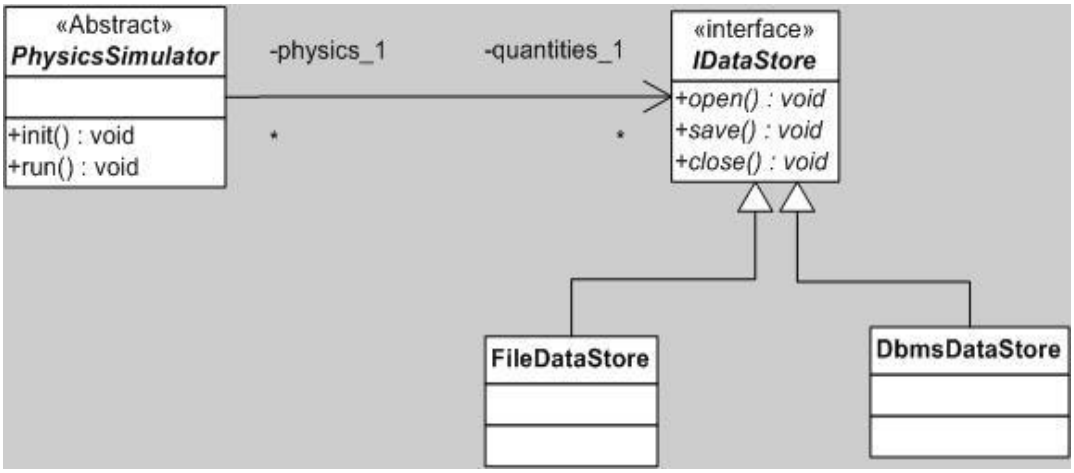
Each interface's names are self-explanatory. The **IPhysicalConfiguration** class implementation is meant to set up the initial conditions of the simulation. The Physics Simulator uses the **IPhysicalAlgorithm** class implementation to numerically integrate the equations governing the simulation. The **IPhysicalMeasurement** class implementation dictates what physical quantities will be periodically measured.

The **IFinalReport** is provided so that summary statistics and final calculations can be reported to the user in a plain text format. The **IFinalReport** is provided so that summary statistics and final calculations can be reported to the user in a plain text format.

The Simulation class is Singleton since there is only one simulation being performed at a time and it is a Bean. class with properties that can be set using a setter method and retrieved using a get method.

The **Marshaller** class is responsible for processes known as marshalling and unmarshalling the Simulation object. The Physics Simulator creates a Simulation from the Simulation.properties file by invoking the marshall() method of the Marshaller class. Based on the values in the properties file, it then sets the values of the properties you specified.

The last package in the framework is the **DataStore** package. The purpose of the classes in this package is to make it transparent to the Physics Simulator class how it saves the data taken during a **IPhysicalMeasurement**.

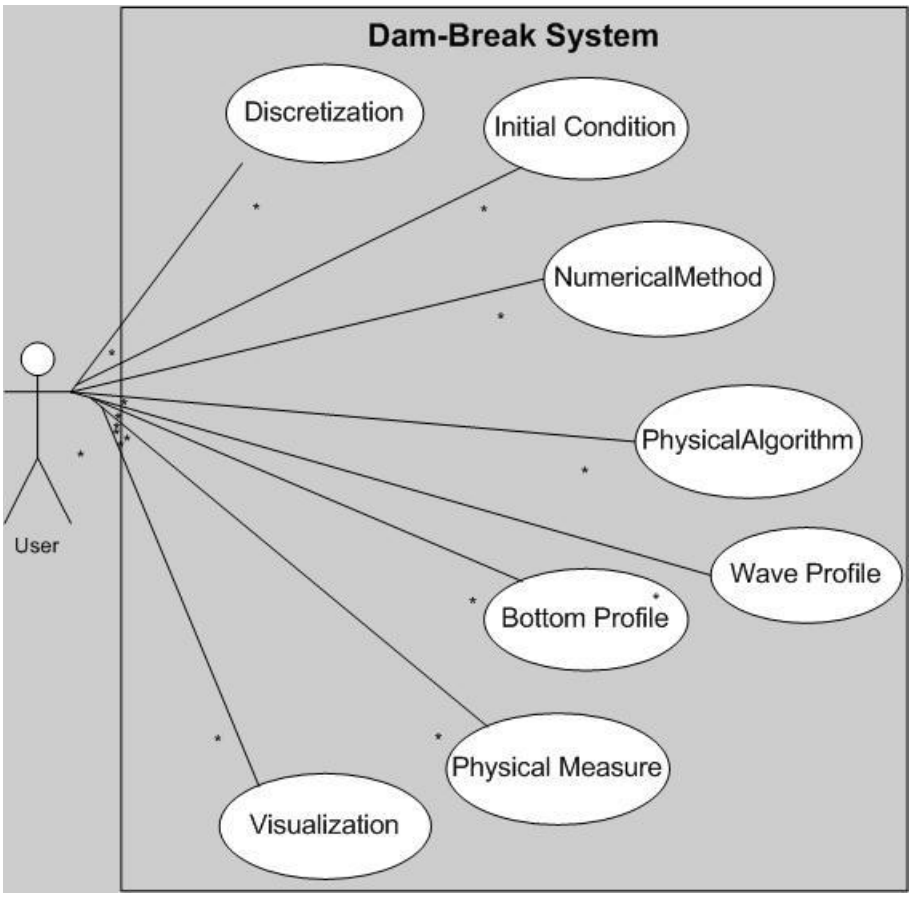


Class Diagram

The abstractions presented above are now expressed in the UML language called class diagram. Central to the UML notation is the concept of class data. A class is an abstract, user-defined description of a type of data. It identifies the attributes of the data and the operations that can be performed on instances (i.e. objects) of the data. This modeling technique allows description of a system using the same terminology as the corresponding real-world objects and their associated characteristics.

Typical scenario

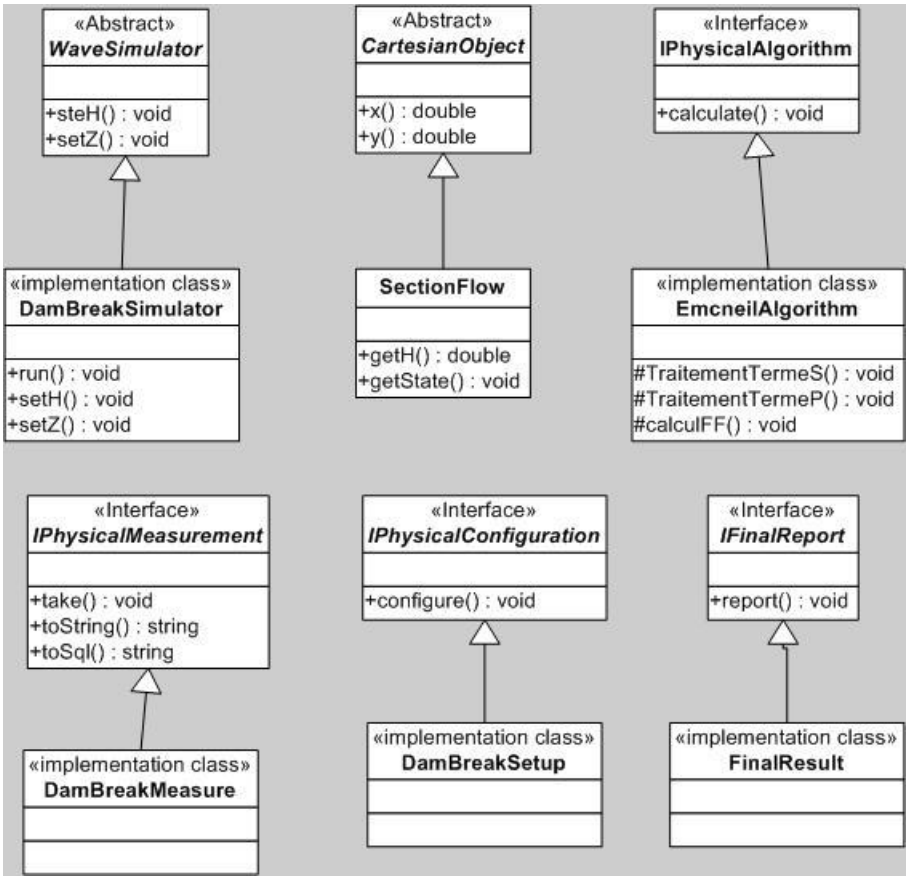
1. Define the discretization
2. Initial Condition
3. The Physical Algorithm (numerical method) needed to perform the Simulation.
4. The Physical Measurement to be performed.
5. A Data Source to store Physical Measurement information.
6. If specified, a Final Report for statistic, visualizing the solution, etc...



Cookbook Recipes

1. The following steps outline how to develop a simulation using the Framework.
2. Create a Physical Simulator by sub-classing the appropriate system object
3. Provide an implementation for IPhysicalConfiguration
4. Provide an implementation for IPhysicalAlgorithm
5. Provide an implementation for IPhysicalMeasurement
6. A Data Source to store physical measurement information
7. If specified, a Final Report for statistic, visualizing the solution, etc…

This is expressed by the following diagram below:



SFX Numeric Types

The library provides types to solve the dam-break problem by solutioning the one-dimensional St-Venant in a conservative form.

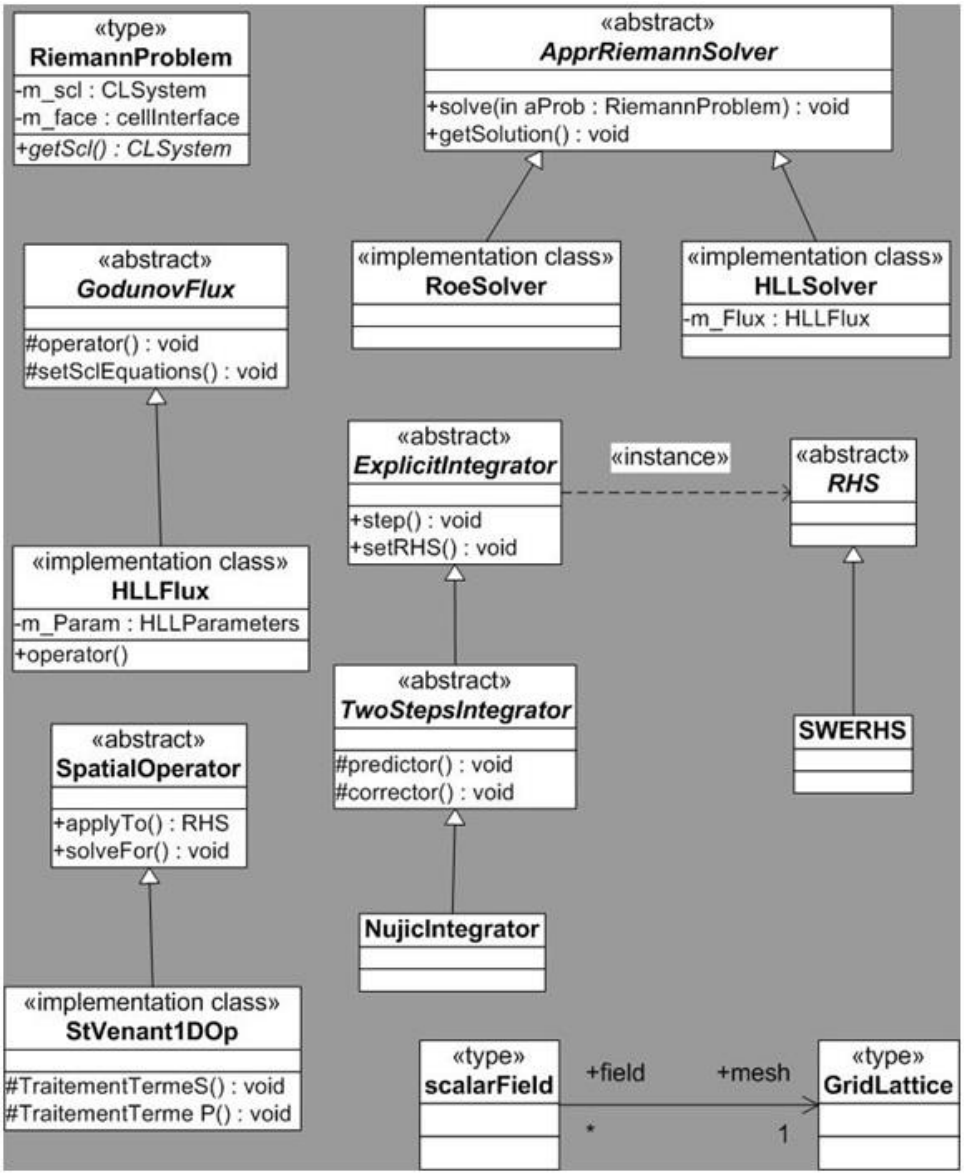
scalarField/GridLattice class (finite difference discretization) Represent the grid (discretization parameters finite difference). This class represents the spatial discretization key concept described above. It holds the parameters of the spatial discretization that will be used by the scalar field, which strongly depend on the discretization.

ExplicitIntegrator Numerical integration (explicit time stepping). Also, time stepping can be achieved by different algorithm, Runge-Kutta of 2nd order, … up to many order approximation (Two-steps family).

Riemann Solver

In the problem that we are interested in, we need to approximate the numerical flux at cell-interface by some algorithm, for example, we can choose to solve it by a Riemann solver, or, we can use some extrapolation method (ENO flux extrapolation). Numerical algorithm to solve the problem taking account the physics, numerical treatment of each term of the equation

Other type of the **DamBreak++** library are presented in the diagram below.


«type»
RiemannProblem
-m_scl : CLSystem
-m_face : cellInterface
+getScl() : CLSystem
«abstract»
ApprRiemannSolver
+solve(in aProb : RiemannProblem) : void
+getSolution() : void
«abstract»
GodunovFlux
#operator() : void
#setSclEquations() : void
«implementation class»
RoeSolver
«implementation class»
HLLSolver
-m_Flux : HLLFlux
«abstract»
ExplicitIntegrator
+step() : void
+setRHS() : void
«instance»
«abstract»
RHS
«implementation class»
HLLFlux
-m_Param : HLLParameters
+operator()
«abstract»
TwoStepsIntegrator
#predictor() : void
#corrector() : void
SWERHS
«abstract»
SpatialOperator
+applyTo() : RHS
+solveFor() : void
NujicIntegrator
«implementation class»
StVenant1DOp
#TraitementTermeS() : void
#TraitementTerme P() : void
«type»
scalarField
+field
+mesh
«type»
GridLattice
*
1