The object oriented design of the integrated Water Modelling System MOHID

F. Braunschweig[a*], P. C. Leitao[b], L. Fernandes[a], P. Pina[a] and R. J. J. Neves[a]

[a]Instituto Superior Tecnico, Technical University of Lisbon,
Av. Rovisco Pais, 1049-001 Lisboa, Portugal

[b]Hidromod Lda., Sala 349, Nucleo Central, Taguspark,
P-2780-920, Oeiras, Portugal

With increasing computer power, the modelling tools for water resources, nowadays, not only integrate physical-based transport models with biogeochemical process models, but also couple surface water body models, groundwater models and hydrographic basin models. This paper describes how the MOHID water modelling system achieves the mentioned integration using object oriented programming in FORTRAN 95. After a short historical overview of MOHID's evolution this paper focuses on the object oriented design of the water modelling system MOHID and the way how object oriented features are implemented in FORTRAN 95. Attention is also given to the way how the numerical software code interacts with the graphical user interface. It follows an exemplification of how the object oriented design is useful for implementing complex system for two cases: the operational model of the Tagus Estuary and the coupling of the Trancao Basin model to the Tagus Estuary model.

1. INTRODUCTION

1.1. History of MOHID

The development of MOHID started back in 1985. Since that time a continuous development effort of new features has been maintained. Model updates and improvements were made available in a regular basis were used in the framework of many research and engineering projects. Initially, MOHID was a two-dimensional tidal model written in FORTRAN 77 [1]. This version also gave the present name to model, which derives from the Portuguese abbreviation of "*MOdelo HIDrodinamico*" (Hydrodynamic Model). This model was used to study estuaries and coastal areas using a classic finite-differences approach. In the subsequent years, two-dimensional eulerian and lagrangian transport modules were included in this model, as well as a Boussinesq model for non-hydrostatic gravity waves [2]. The first three-dimensional version of the model was introduced with the version MOHID 3D which used a vertical double Sigma coordinate [3]. The limitations of the double Sigma coordinate revealed the necessity to develop a new version

*This work was carried out within the tempQsim research project funded by the European Union under contract n EVK1-CT-2002-00112

which could use a generic vertical coordinate, permitting the user to choose from several coordinates, depending of the main processes in the study area. This necessity led to the introduction of the concept of the finite volumes approach which was introduced in the version MESH 3D [4]. In the MESH 3D model, a 3D eulerian transport model, a 3D lagrangian transport model [5] and a zero-dimensional water quality model [6] were included. This version revealed that the use of an integrated model based on a generic vertical coordinate is a very powerful tool. However it was verified that the model was difficult to maintain and to extend due to the FORTRAN 77 language limitations and due to the increasing number of users and programmers and the interdisciplinary character of the modelled processes. Thus, it was necessary to establish a methodology which permitted to reuse the code more often and improve its robustness related to programming errors [7]. It was decided to reorganize the model, writing it in ANSI FORTRAN 95, profiting from all its new features, including the ability to produce object oriented programming with it, although it is not an object oriented language. This migration began in 1998, implementing object oriented features like those described in Decyk [8] with significant changes in code organization [9]. This migration resulted in an object oriented model for surface water bodies which integrates scales and processes [7].

1.2. Present State

The object oriented modular design of this model was the base to build the present MOHID Water Modelling System, which is a set of several numerical tools. Presently there are three core tools: (i) MOHID Water, (ii) MOHID Land and (iii) MOHID Soil. The first tool is the improved version of the model described above; the second one is a watershed model and the third one simulates water flow through porous media. Today the MOHID Water Modelling System is a reliable and robust framework for the upcoming challenges of water modelling. MOHID Water's versatility can easily be demonstrated by the range of applications carried out in the last couple of years: applications in the North Atlantic to study general circulation [10,11], Oil Spills [12], eutrophication and residence times in estuaries [13] and reservoirs [14]. MOHID Land is the most recent core executable of the MOHID Framework and doesn't have yet the maturity of MOHID Water. So far it was applied to some watersheds, among them the Trancao Basin, which outlet is the Tagus Estuary.

2. OBJECT ORIENTED DESIGN OF MOHID

2.1. General

This chapter describes how object oriented features are implemented in the MOHID water modelling system. In common object oriented languages (OOL), like JAVA or VB.NET, objects are created from classes. FORTRAN 95 is not an OOL, but it can be used for object oriented programming (OOP). FORTRAN modules can act as classes of common OOL [8,15]. As mentioned before, presently the MOHID Water Modelling System is written in FORTRAN 95. The whole structure of the system is divided into FORTRAN modules, each of them having the functionality of an object class. MOHID's design uses several object oriented features like encapsulation, polymorphism, function overloading and inheritance. Objects have four standard methods: (i) constructor, (ii) selector, (iii) modifier and (iv) destructor.

```
module Class

    implicit none

   private
```

Figure 1. Example of a class definition and encapsulation using the MODULE and PRIVATE statement

Some MOHID classes from different levels are:

EnterData Parses ASCII data files written in format similar to XML and extracts information. Low level class used by all modules which need to read data files.

Public information: Any information extracted from the data file.

HorizontalGrid Handles information over a structured regular horizontal grid. Intermediate class used by all three MOHID core executables to implement the grid they work over.

Public information: Grid rotation, areas of grid cells, coordinates of cell centers, coordinates of cell vertices and distances between centers of cells.

Hydrodynamic Solves the non-turbulent flow properties of the water column of surface water bodies (e.g. oceans, estuaries, reservoirs). High level class just incorporated in MOHID Water.

Public information: Water level, velocity field and water fluxes among grid cells.

2.2. A standard MOHID Class

Each of the more than 50 classes that form the MOHID Framework is designed to fill out some standard requirements, regarding programming rules and definition concepts, in order to establish a straightforward connection of the whole code. This standardization is reflected in memory organization, public methods systematization, possible object states, client/server relationship and errors management [7]. Each class is responsible for managing a specific kind of information. The design of a class, in FORTRAN 95, can be accomplished by the MODULE statement. This way, information can be encapsulated using the PRIVATE statement (Figure 1). Encapsulation assures that all the information associated to an object is only changed by the object itself, reducing errors due to careless information handling in other classes.

A MOHID class is defined as a derived type, which has, in addition its specific information, two required variables (Figure 2): *InstanceID* and *Next*. *InstanceID* relates to the identification number of the class instance, that is the object's ID, which is attributed when the object is created.

Each time a new object is created, it is added to a collection of objects, stored in a linked list. *Next* relates to the object stored after the current in the list. The linked list

```
private :: T_Class
type       T_Class
    integer                         :: InstanceID
    type(T_Class), pointer          :: Next
end type   T_Class
```

Figure 2. Example of a MOHID class derived type definition

```
type (T_Class), pointer          :: FirstObject
type (T_Class), pointer          :: Me
```

Figure 3. Global class variables

is designed to be one-way, that is, it can only be scanned in one direction, because there was no need to turn it more complex (two-way or four-way) and it would only require more allocated memory.

Each class has only two global variables, defined as derived type pointers (Figure 3). They are the first object in the linked list (*FirstObject*), which works as an anchor or starting point to scan the list of instances of the module, and a pointer to the current active object (*Me*). The procedure to access an object is to, starting on the first object, scan the list and find the corresponding one through its ID number. Each module contains its own method to perform this scan, which is always called in the beginning of any public standard method (beside the constructors, since there the object is not created yet). After successfully locating an instance of a module in the linked list, the global variable *Me* points to the current instance.

2.2.1. Object States

A MOHID object can have two primary states: ON and OFF, standing for if the object has been constructed or not. In order to create a new object, the client object must use a public constructor method which set its state to ON. If a client object tries to access memory of an object that has not been constructed, therefore it does not exist, an error message is returned, which normally leads to stop execution. If an object is ON it can have two secondary states: READ-LOCK or IDLE. If an object is READ-LOCK it means that one or more client objects are accessing information, but without changing it. During this state, no public methods that lead to information alteration can be invoked. Each time a user stops reading, it invokes the READ-UNLOCK method, which removes one reader from the readers list. If no client objects are accessing information, then the object state is set to IDLE. This means although it exists, it is inactive. In order to read information

from an object, the object must be IDLE or it must be READ-LOCK, as more than one reader is allowed. In previous versions an object could also have the WRITE-LOCK state, but this feature was removed from the code as it was somehow redundant. The WRITE-LOCK state related to the phase when a client object modifies the object's information. This meant that other objects could not access or modify the information, once the object is in a transition phase. This state was first conceived to perform parallel processing, as an object could not be read if it was still being modified, leading the program execution to wait and improving robustness in accessing memory. Although wise, this feature has proven to be unnecessary, as in all the code, never an object's public method was invoked when an object was WRITE-LOCK, because the locking and unlocking of this state was performed at the beginning and end of every public modifier method. This way, in order to be modified, an object must be IDLE, i.e. it must be ON but inactive, waiting for instructions. Whenever one is invoking public methods and an inconsistency occurs in the client/server communication, a message is returned by the server indicating the type of error. Using this error message the client then decides the action to take: whether to continue without warning the user or send him a warning message and let him decide what to do, or stop execution if the error message compromises the program continuity.

2.2.2. Object Collector

Objects are only created once and the ID number of any new created object (server object) is returned to the calling (client) object. In order to a second client access information on the same server object, the server object's ID must be provided by the first client object. In MOHID Framework this is called association. This association is managed by the Object Collector. The Object Collector is a derived type array, where in each array position information is stored about the corresponding class instance. This information relates to the ID number of an object (*InstanceID*), the number of client objects associated to it (USERS) i.e. that can have access to it, the number of client objects reading information from it (READERS) and the object state (READ-LOCK). The object collector is stored in a module which is used by all other modules of the MOHID Framework. There are several public methods which can be accessed to modify the object collector: (i) one method to register the instance of any new created object, (ii) methods increment/decrement the number of users of an object and (iii) methods to get/set the READ-LOCK state. The ON/OFF state of a module can be inquired through the number of user.

2.2.3. Constructor Methods

Constructor methods are used to construct new instances of classes. There are several standard tasks which are performed by all modules of the MOHID Framework, when they pass through the constructor method. First of all the new instance is created and registered in the object collector. Afterwards it's verified if the object state is off, so it's guaranteed that an object is not created twice. In case the state is off, the memory of the instance is allocated and the object added to the module's linked list of objects. After this all other constructor logic is performed (e.g. allocation and initialization of state variables) and the instance ID is returned to the client object.

2.2.4. Modifier Methods

Modifiers methods are used to modify the state variables of an object. When they are called, the correct instance of the module is located in the linked list through the instance ID, which is received from the client module by argument.

2.2.5. Selector Methods

Selector methods are used to access encapsulated information of an object. All selector methods in MOHID start with *Get*. Object location within the linked list is performed in the same way as in the modifiers methods. After successfully locating the instance of a module, the selector method returns the desired information. For performance reasons, in the case of matrixes, the selector methods return pointer arrays. In this case state of the object providing the information is set to READ-LOCK, so it's protected against modification, once its information is accessed from outside of the module. The state turns IDLE again if the client module releases the pointer array by calling an *UnGet* method.

2.2.6. Destructor Methods

Destructor methods are used to remove an object from the modules linked objects lists. Like the Modifier and the Selector methods, the destructor methods receive the instance ID by argument from the client. After successfully locating the object, the memory used by the object is deallocated. Afterwards the object is removed from the module linked objects list.

2.2.7. Objects inside modules

Besides creating objects from FORTRAN modules, MOHID also implements objects within modules. The way this kind of objects are created follow the same logic, with the difference that these are not directly accessible from outside the module, therefore they don't have the READ-LOCK state. Also they are not registered in the *ObjectCollector*, so they can not be inherited which since they are private, also would not be possible. This way of object creation for this kind of object is always used for ”smaller” or individual objects. For example, MOHID's class Discharges handles a conjunction of discharges (For example all rivers which discharge into an estuary). Inside class Discharges, individual discharges and their properties (e.g. geographical location, flow evolution, property concentrations) are stored in a linked list of type *IndividualDischarge*.

2.3. MOHID Objects

MOHID's modules are structured in a hierarchical way. All executables of the MOHID Water Modelling System are built on the top of one or more base libraries. The hierarchical structure of MOHID's Framework is presented in Figure 4.

The three core executables files (MOHID Water, MOHID Soil and MOHID Land) can be found at the top of the pyramid. Smaller utility programs are easily built on the top of the libraries, which are usually designed for pre or post processing results of the core executables. The advantage of the object oriented design in a complex system like MOHID, is easily shown by a class like *EnterData*. This class is a very low level class (it can be found in MOHID Base 1) and its purpose is to read MOHID ASCII data files (see below). Almost every module which handles any kind of process uses a separate instance of module *EnterData*, since all modules have to read a specific data file. Once a powerful

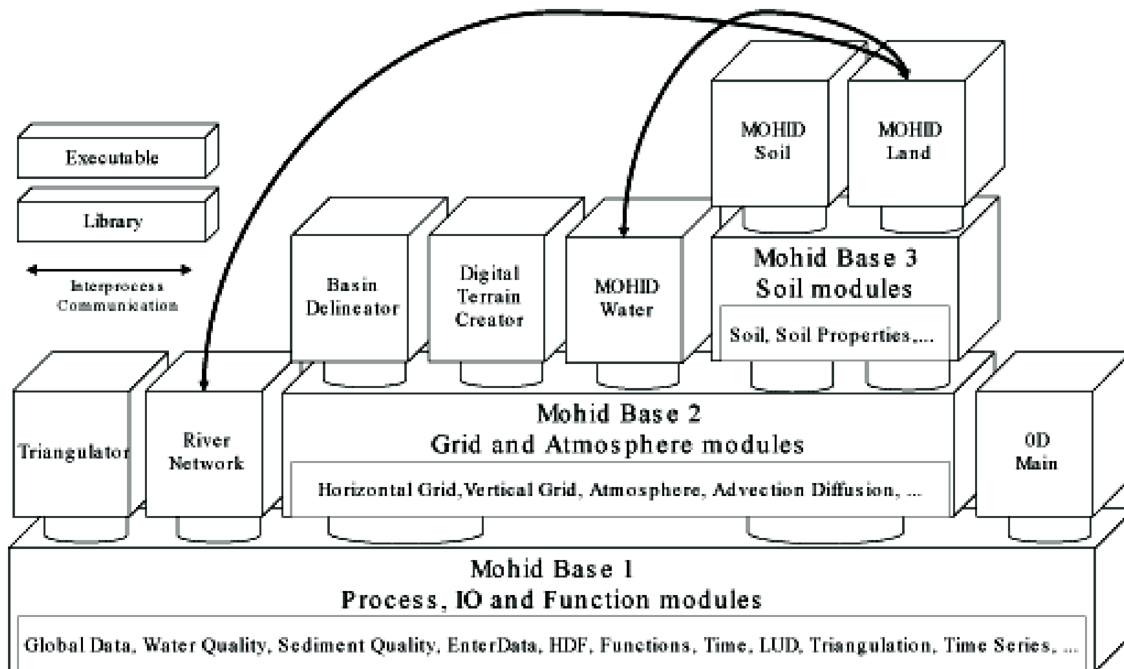

Figure 4. MOHID Framework, the hierarchical design

class like the class *EnterData* is build, the functionalities can be accessed by all modules which inherit (by the USE statement) the public methods from this class. Protection of any data (e.g. the unit number) is guaranteed by encapsulation. Public methods take care of opening, reading and closing the data files. Selectors are available to read data in any specific way. Once a feature is added to this class, it becomes available to all other classes which "use" module *EnterData*. Other classes belong exclusively to one executable and rarely more than one instance of them is created. For example, the class Hydrodynamic, which solves the 3D flow field over a given geometry, is just compiled with the MOHID Water executable. More than one instance of this class is only created when nested models are set up. On the top of MOHID Water exists class *Model* responsible to coordinate the evolution of one surface water body:

1. temporal evolution and fluxes between the three compartments (atmosphere , water column and sediments);

2. temporal evolution inside each compartment (e.g. the evolution of the water column and information exchange between three core classes: *Hydrodynamic*, *Turbulence* and *Waterproperties*);

3. the information exchange between models and sub-models;

4. the temporal evolution of the model;

2.4. User Interface General

2.4.1. General

As numerical tools like MOHID become more complex, the need for a well structured graphical user interface grows. With the development of the MOHID Framework a graphical user interface was built in parallel. In former times the interface was built in Visual FORTRAN, as a tool on the top of the MOHID's base libraries, but it turned out that the design of a simple windows interface is very demanding to program, in comparison with languages which are designed for this purpose, like JAVA or Visual Basic. In the last year it was decided to develop a new graphical user interface on the .NET platform. This option revealed itself to be a positive one, as programming and development efficiency increased significantly. Today MOHID's graphical user interface is composed by two core programs: (i) MOHID GUI which handles the directory structure and data files necessary to set up a set of MOHID simulations and (ii) MOHID GIS which is a tool which handles geo-referenced data and visualizes results produced by MOHID. It's written mainly in VB.NET and so it uses full object oriented features. Information exchange between the numerical tools and the graphical user interface is achieved by two file formats: (i) the format implemented by the class *EnterData* and (ii) the Hierarchical Data Format (HDF).

2.4.2. Information Exchange

MOHID's class *EnterData* allows to write and read ASCII data files structured in a similar way as XML files. The major reason why this class doesn't read XML files is that it was implemented just before XML turned popular. It would be quiet easy to modify this class to read XML files, but the backward compatibility with all existing data files would be lost. In order to exchange information between the numerical tools written in FORTRAN 95 and the graphical user interface written in .NET class *EnterData* has been developed in both languages. Another way to change information between the graphical user interface and the numerical tools is the class HDF. Like the class *EnterData* this class is implemented in both languages. MOHID's HDF class is a class on the top of the HDF library. This library is developed and maintained by the National Center for Supercomputing Applications. The main functionality of this library is to store matrix data in a structured way [16].

3. IMPLEMENTATION

This chapter describes the advantages of design a system like MOHID using object oriented programming over models which are written using sequential programming. This approach is very common in FORTRAN programming community. To describe these advantages, two examples are presented which show in which way the object oriented approach is beneficial: (i) one for implementing nested models of MOHID Water and (ii) one for joining watershed models with estuary models.

3.1. Nested Models

The Tagus Operational Model implements several models of MOHID Water in three nested levels. The largest level covers almost the entire western Portuguese coast and its functionality is to supply boundary conditions to nested models of level 2. Models of the second level cover the whole Tagus Estuary and are used for two proposes: (i)

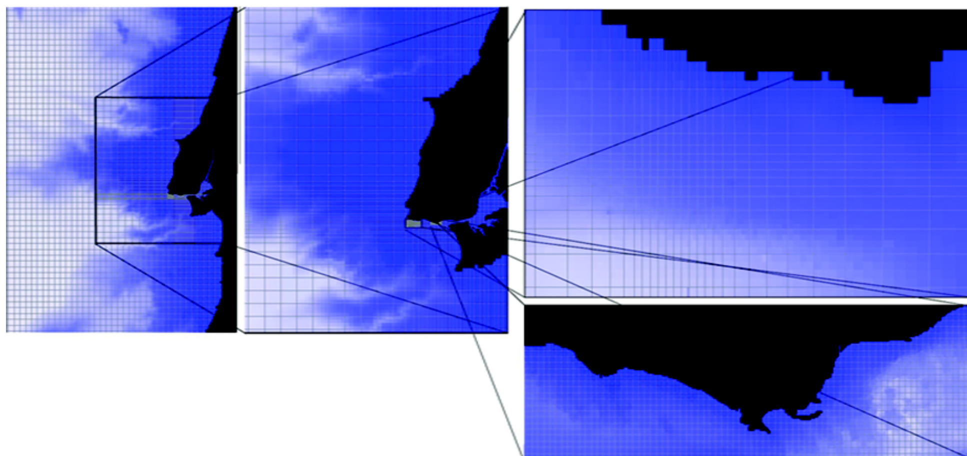

Figure 5. Three levels of nested models. Different models have different color scales. From left to right images show large part of the Portuguese coast, Tagus Estuary, Guia Outfall region (top), Caracavelos Beach (bottom).

to simulate flow, transport and ecologic processes inside the estuary and (ii) to supply boundary condition for nested models of level three. Level three models are models at local scale, set up to study water quality problems close to beaches or to monitor the impact of submarine outfall. They can be coupled as needed. Figure 5 shows the nesting of these models.

There are two possible ways to run such simulations:

1. Run a single executable which simulates the three levels, one instance of class Model for each model instance in the different level. If for example level three would have 2 models, in total, four different instances of class *Model* would be created.

2. Run four different executable, each one simulating one model. Inter-model communication is achieved by communication with the Message Passing Interface (MPI).

The first way is a good example of how the object oriented design, like the one used within the MOHID Framework, can be useful to set up nested models. Since there is no limitation of creating instances of individual classes, a model with infinite number of sub-models can be set up in order to study processes in a desired detail. The limitation found, during the implementation of the operational model of the Tagus Estuary, was the available computer power. Running several models on a single processor turned execution time into the limiting factor. To overcome this limitation, parallel processing for different models was implemented, like described in 2.

3.2. Watershed Modelling

MOHID Land is the newest core executable of the MOHID Water Modelling System and still to improve. This chapter describes how the object oriented design of the MOHID

Framework enabled a quick development of this tool. Many existing classes of the MOHID Framework could be used to build MOHID Land:

1. classes related to input/output like *EnterData*, *HDF* and *TimeSeries*;
2. classes related to the underlying grid like *HorizontalGrid*, *HorizontalMap* and *GridData*;
3. classes which impose boundary conditions like *Discharges* and *Atmosphere*;
4. zero dimensional processes classes (e.g. classes which simulate the dynamics of nutrients inside the water column)

To some of these classes some functionality was added. For example, in former versions class *GridData* was just used to relate bathymetric data with class *HorizontalGrid*. These class was modified in a way that any spatial variable data can be related with the underlying grid (class *GridData* inherits class *HorizontalGrid*). MOHID Land uses class *GridData* not only to obtain information about topography, but also for land use, soil type, etc. New classes developed are related with specific processes which occur inside a watershed. Examples are:

1. class *Runoff* which calculates overland runoff;
2. class *Infiltration* which handles infiltration processes;
3. class *DrainageNetwork* which handles water routing inside rivers;

3.2.1. Tool Integration

The integration of different tools, namely MOHID Water and MOHID Land can be used to study the water cycle in an integrated approach. Since these two tools are based on the same framework, the coupling of them is easily achieved. One way of coupling these two models is to run MOHID Land first for all basins which discharge to a water body and create a time series at the outlet of the watershed. Afterwards this time series are used to impose the boundary conditions for MOHID Water. Since both models use the class *TimeSerie* the integration of these two models in this way was immediate. The problem of this integration is that MOHID Land can never be influenced by MOHID Water, but there are some cases where this influence might be important:

1. If the downstream region of a river simulated by MOHID Land is influenced by tide, the river class *DrainageNetwork* needs the downstream boundary condition from a model like MOHID Water. This condition occurs also at Trancao River (Figure 6).

2. A network of reservoirs, linked to agriculture industries is a typical case where a model like MOHID Land would need feedback from a model like MOHID Water in "real-time". To overcome these problems, the coupling of these two models by the Message Passing Interface is under study. In this way updated information can be exchanged in every time step.

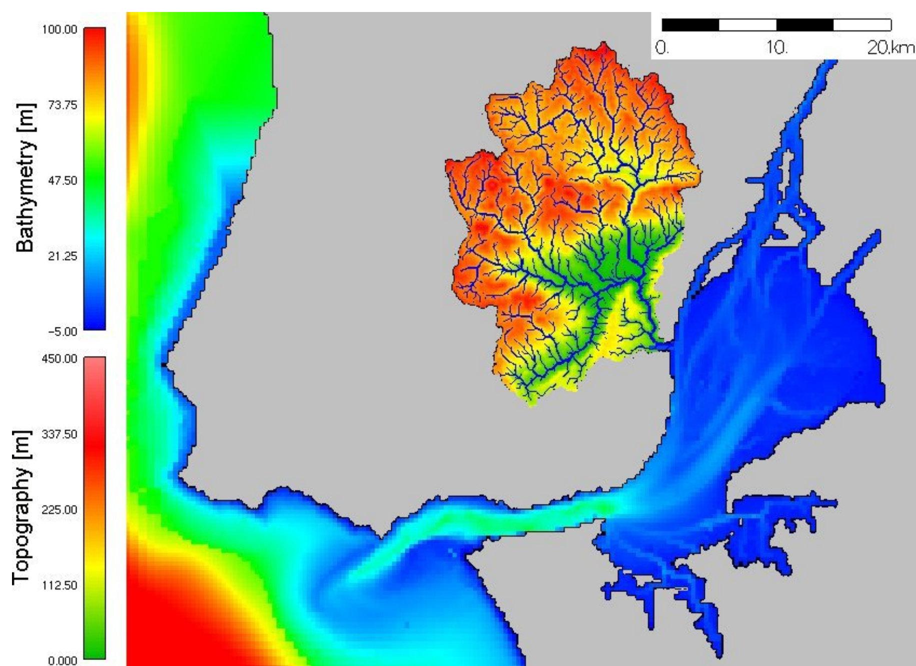

Figure 6. The integrated Trancao Basin / Tagus Estuary System

4. CONCLUSIONS

The way an object oriented design for an integrated water modelling system like MOHID can be beneficial is presented. The use of object oriented programming features allows MOHID Framework to grow to a system which nowadays has not only an increasing number of users, but also software development contributors. Since MOHID uses the object oriented design, errors related to bad memory management (which where frequent in older versions) disappeared completely. The common design of all MOHID classes, related to encapsulation/public methods not only turned the code more reliable and protected against errors, but also led to a very well structured code, easily understood by new scientists that work with the model, even taken into account the over 200k code lines which compose MOHID Framework. A major drawback of the object oriented design, not focused in the article, is the increase in execution time of the model. Former comparisons showed the first object oriented version of MOHID need two to three times more CPU time then the last version of this model in FORTRAN 77 [9]. To gain reliability, MOHID Land must be applied to several watersheds. This will hopefully happen during ongoing scientific and engineering projects where the MOHID Water Modelling System will be used. To close the water cycle, a groundwater model must be coupled to MOHID. The development of a tool of this kind is in progress.

REFERENCES

1. R. J. J. Neves, Etude Experimentale et Modelisation des Circulations Trasitoire et Residuelle dans l'Estuaire du Sado. Ph. D. Thesis, University de Liege (1985)
2. A. J. R. Silva, Modelaao Matematica Nao Linear de Ondas de Superficie e de Correntes Litorais. Ph. D. Thesis, Technical University of Lisbon (1991)
3. A. J. Santos, Modelo Hidrodinamico Tridimensional de Circulacao Oceanica e Estuarina. Ph. D. Thesis, Technical University of Lisbon (1995)
4. F. Martins, Modelacao Matematica Tridimensional de Escoamentos Costeiros e Estuarinos usando uma Abordagem de Coordenada Vertical Generica. Ph. D. Thesis, Technical University of Lisbon (1999)
5. P. C. Leitao, Modelo de Dispersao Lagrangeano Tridimensional. M. Sc. Thesis, Technical University of Lisbon (1996)
6. R. Miranda, Nitrogen Biogeochemical Cycle Modeling in the North Atlantic Ocean. M. Sc. Thesis, Technical University of Lisbon (1999)
7. P. C. Leitao, Integracao de Escalas e Processos na Modelacao do Ambiente Marinho, Ph. D. Thesis, Technical University of Lisbon (2003)
8. V. K. Decyk, C. D. Norton and B. K. Szymanski, Expressing Object-Oriented Concepts in Fortran90. ACM Fortran Forum, Vol. 16 (1997)
9. R. Miranda, F. Braunschweig, P. C. Leitao, R. J. J. Neves, F. Martins and A. Santos, Mohid 2000, A Coastal integrated object oriented model. Hydraulic Engineering Software VIII, WIT Press (2000)
10. R. J. J. Neves, H. Coelho, P. C. Leitao, H. Martins and A. Santos, A numerical investigation of the slope current along the western European margin. Computational Methods in Water Resources XII, 2, 369-376, (1998)
11. H. S. Coelho, R. J. J. Neves, M. White, P. C. Leitao and A. J. Santos, A model for ocean circulation on the Iberian coast. Journal of Marine Systems. Vol. 32 153- 179. (1997)
12. L. C. Leitao, P. C. Leitao, F. Braunschweig, R. Fernandes, R. J. J. Neves and P. Montero, Emergency activities support by an operational forecast system - The Prestige accident. 4th Seminar of the Marine Environment, Rio de Janeiro (2003)
13. F. Braunschweig, F. Martins, P. C. Leitao and R. J. J. Neves A methodology to estimate renewal time scales in estuaries: the Tagus Estuary case, Ocean Dynamics, Vol. 53, N3, 137-145. (2003)
14. F. Braunschweig, Generalizacao de um modelo de circulacao costeira para albufeiras, M. Sc. Thesis, Technical University of Lisbon (2001)
15. J. E. Akin, Object Oriented Programming via FORTRAN 90, Engineering Computations, v. 16, n. 1, pp. 26-48 (1999)
16. M. Folk, Introduction to HDF5, NCSA/University of Illinois at Urbana-Champaign http://hdf.ncsa.uiuc.edu/HDF5/papers (2000)