JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 6, November-December 2003

E-MOBI Smart Object Model and Implementation

Gerardo Rossel and Andrea Manna, Department of Computer Science, University of Buenos Aires, Argentina

Abstract

MOBI (Object Intelligent Model) is a multi-paradigm model for the incorporation of rule-based processing and knowledge to object-oriented languages. MOBI approach is a generic software architecture capable of being implemented in object-oriented languages. The nature of this architecture allows to add knowledge to instances and classify the knowledge in the class hierarchy, achieving the definition of a concept of knowledge inheritance. The purpose of this paper is to show the MOBI's Eiffel implementation named E-MOBI. This implementation supports multiple knowledge inheritance and the main features of MOBI's architecture. We will show that E-MOBI is one proper tool for the creation of intelligent agents for Internet. At end, a case study is presented to show the flexibility for the implementation of intelligent agents.

1 MOBI OBJECT INTELLIGENT MODEL

MOBI (Modelo de OBjetos Inteligentes in Spanish) [Man 01] is a multi-paradigm generic architecture that permits to incorporate facilities of programming in logic to an Object Oriented language. MOBI is founded in three basic premises:

- Do not incorporate new syntactical structures to language
- Provide an extension as part of the class libraries
- Build towards class-based object-oriented languages

With these points we tried to obtain a model which emphasizes simplicity and portability. Without incorporating syntactical structures to the language the needs of a precompiler or a modified compiler is avoided. If the extension is standard enough, it is able to operate with only a few modifications thru different language implementations. The model is quite wide to be adapted to any class-based object-oriented language. The instance-based languages (eg. Self [Ung 87].) require a special consideration and are not still contemplated in MOBI.

Cite this article as follows: Gerardo Rossel, Andrea Manna: "E-MOBI Smart Object Model and Implementation", in *Journal of Object Technology*, vol. 2, no. 6, November-December 2003, pp. 103-115. http://www.jot.fm/issues/issue_2003_11/article3

MOBI encloses the following characteristics:

1. The classes have a common Knowledge Base for all their instances.
2. It is not necessary that every class in the domain problem has a Knowledge Base
3. The classes having a Knowledge Base must include a common interface to access the Base.
4. The objects (instances) can have a private and dynamic Knowledge Base that can be different for objects belonging to the same class.
5. Behavior:
    a. The methods, which are defined in classes, can consult the knowledge base of the regarding class.
    b. The methods can be programmed in a completely imperative form or can, in the method body, consult their Knowledge Base and act accordingly.
6. The internal state of an object in a determined execution point of a program, is defined by the values contained in their instance variables (or structural attributes) and the knowledge that it has.
7. The knowledge is inherited through the inheritance hierarchy.
8. The objects can transfer their private knowledge to the class they belong to share it with the remaining instances of the class.

These previous items define the computational model of MOBI. The main components of a MOBI application are: Knowledge Base, Support Algorithms and Domain's Classes. Their relation are shown in Fig. 1.

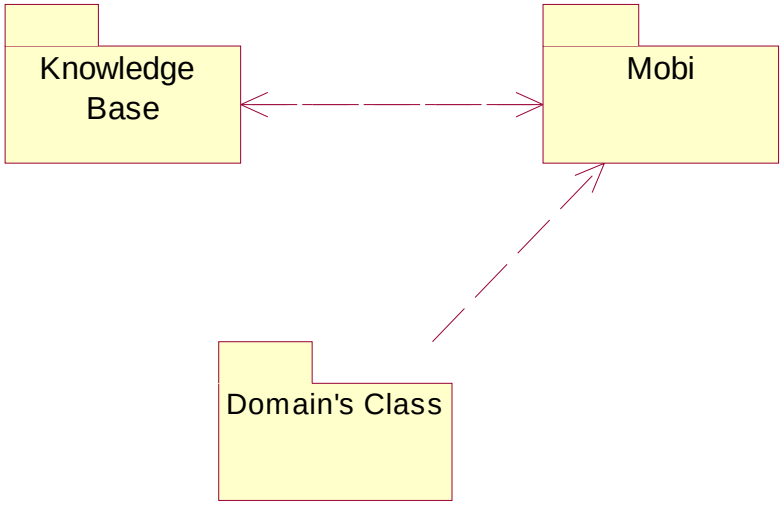
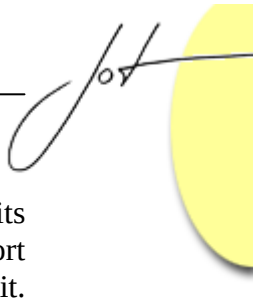


Fig. 1 Main Components



The interaction with the Knowledge Base and the association between a class and its Knowledge Base only takes place by means of the support algorithms. The support algorithms allow a class to access its Knowledge Base and to make inferences from it. The inference algorithm proposed by MOBI is SLD-Resolution. The SLD-Resolution is a refinement strategy of Robinson's resolution principle [Ron 65], applied only to defined clauses. The utilization of SLD-Resolution obeys to the settle of a correct and complete inference mechanism [Llo 87]. It provides a foundation for a logically sound operational semantic of defined programs.The support algorithms are implemented in a library of classes and, in addition, allow to define the necessary interface for the classes that support knowledge. MOBI has a class with the same name that defines the interface of the classes that support knowledge and the operations for the manipulation of the Knowledge Base. It is necessary to make compatible interfaces, which requires a design solution. In MOBI, is suggested the use of the structural Design Pattern Adapter or Wrapper [Gam 95], whose purpose is to turn the interface from a class to other that the clients hope. For both models, Object Adapter and Class Adapter, each MOBI implementation can choose the most recommended for the language. For example, Class Adapter can be used taking advantage of the Eiffel language multiple inheritance. Fig. 2 outlines the way to use Class Adapter in MOBI.

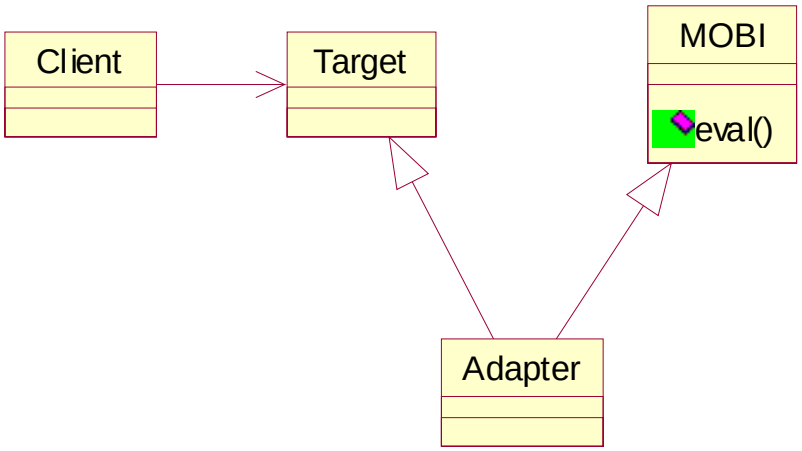


Fig. 2 Class Adapter in MOBI

In MOBI the internal state of an object is determined by its structural attributes and the knowledge acquired throughout its life. The operations to manipulate the Knowledge Base are those that allow the variation of the object Knowledge Base and the class Knowledge Base. On the other hand, MOBI has two ways to use the Knowledge Base:

- *External*: Another object makes a question sending a message or invocation like: *target.evaluate(goal)* which launches the evaluation algorithm on the private and common Knowledge Base.

- *Internal:* An object can, in some method code, invoke an evaluation to act accordingly.

The private Knowledge Base influences so much the internal state as the behavior, while the common Knowledge Base influences only the behavior, since it is shared for all the instances and does not mark the internal state of an individual instance. When the private knowledge is transformed into common knowledge, cease to determine the internal state of the object that generated the knowledge and becomes a common characteristic for all the instances.

2 E-MOBI

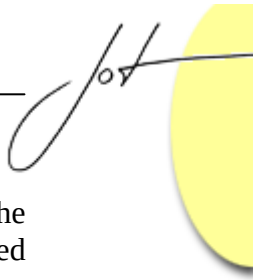
E-mobi is an implementation of MOBI made in Eiffel language [Mey 92]. The language was chosen because of several capabilities that justify its selection, that is to say:

- Pure Object Oriented.
- Strong typing.
- Support of Multiple Inheritance.
- Great amount of class libraries.
- Support of Design by Contract.
- There are no remarkable antecedents for the incorporation of inference mechanisms.

Clearly the last point tube a decisive weight in the selection. The implementation tries to respect the essence of the model, that is to say:

- Knowledge based on rules.
- Private knowledge for the instances.
- Knowledge Inheritance.
- Possibility of defining on him the operations of update of knowledge bases.

One alternative for E-MOBI was to use an external inference machine, but we decided not to use this option in favor of a pure implementation within the own language. We also tried to obtain a flexible design in such a form that the support algorithms (unification and evaluation) could be optimized without affecting the model. In addition we determined to implement the knowledge base using the native Eiffel tools for persistence, instead of some other viable option, for example the use of a database (relational or object-oriented). This last decision had the purpose to avoid jeopardizing the implementation with external tools to the language choice.



As a design decision we solved to create a cluster named MOBI including all the classes implementing the model and present this cluster under the form of a precompiled library. The objective of this decision was to build a reusable tool for any project, where it is only necessary to import the cluster MOBI to be able to generate Knowledge Base classes and intelligent objects.

Clusters are groups of classes logically related. The only constraint is that two classes in a cluster cannot have the same name. Informally the following guides should be followed [Mey 92]:

- Classes in a cluster should be conceptually related.
- The amount of classes in cluster is due to limit depending on the connection between the classes.
- The cycles in the client relation usually involve classes belonging to the same cluster.

In Eiffel the short form (also named **contract form**) is one of the ways to document clusters and classes, and consist in the description of the class interface and its corresponding assertions. We will use this form when describing E-Mobi main classes.

Figure 3 shows a UML diagram with the main relations and classes of the cluster MOBI.

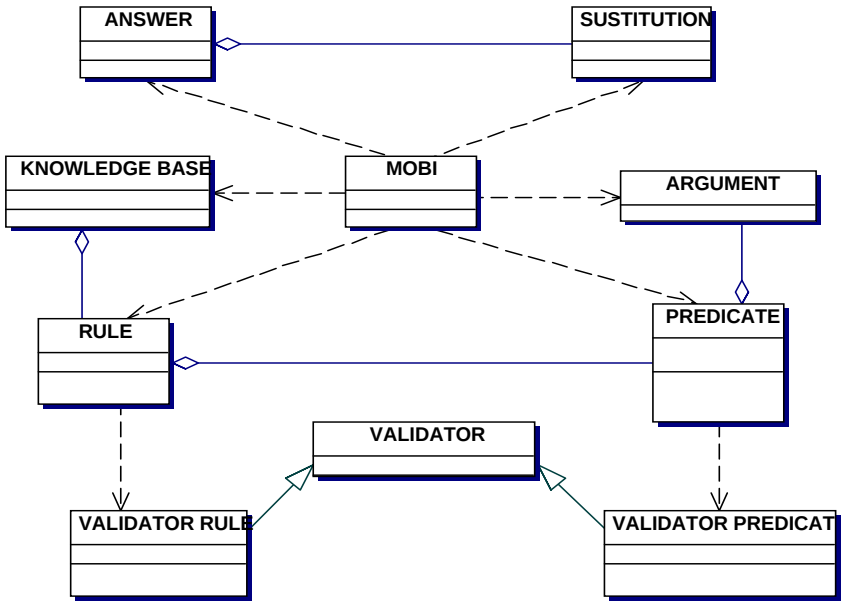


Fig.3 Cluster MOBI

The core of E-mobi implementation is the class named `MOBI`, which counts on the structure definitions and the necessary programming algorithms for using E-mobi. This

class implements the main algorithms of the model (eg. evaluation algorithm, unification algorithm). Next is the contract format of class MOBI:

```
class interface MOBI

inherit

   RTS_SERVER
        export  {NONE} all
      end

feature -- Operations

   eval (sgoal: STRING): ANSWER
         -- Main call to evaluate a predicate on the KB
      require
         ((sgoal) /= (void)) and then (isPredicate (sgoal))

   all_answers (sgoal: STRING): ARRAY [ANSWER]
 --call to obtain all the possible true answers
-- if there is no answer the Array is empty
      require
         ((sgoal) /= (void)) and then (is_predicate (sgoal))

   re_eval: ANSWER
         --  like to semicolon in Prolog,
         --  it requests another answer

   is_predicate(p: STRING): BOOLEAN
-- it returns true if p conforms to the syntax established for
-- predicates

   name_kb: STRING
         --file's name of Knowledge Base

   true_answer : ANSWER
   --Routine ONCE that returns a answer with the value in True

   false_answer: ANSWER
        --Routine ONCE that returns a answer with the
        --value in False

feature -- Operations with Knowledge Base

   save_kb

   expand (r: RULE)
        -- To expand the Knowledge Base with the rule
        -- r is added to the Knowledge Base
```

```
    contract (r: RULE)
          -- To contract the Knowledge Base with rule r
          -- r takes off of the Knowledge Base

    consolidate
          --  It adds to the Class Knowledge Base the
         -- instance knowledge

invariant
   knowledgeprivate_necesary: (kb_private) /= (Void)
end interface -- class MOBI
```

The main routines provided by MOBI are: `eval`, `all_answers` and `re_eval`. The first receives a goal and returns an answer, evaluated to true or false, and, if it is appropriate, the list of substitutions found.

The routine `is_predicate`, permits to verify that a given string has the syntax of a predicate. According to the design by contract ([Mey 97], [Jez 99], [Mit 02]), the client is responsible to check that the goal to evaluate be really a predicate. It is used by precondition of `eval` routine.

An invocation to `eval` returns the first answer that the evaluation algorithm finds. It is possible another answer be needed, whether to know if there is more than one, or because the answer obtained is not acceptable for the context in which was requested. To achieve this, it was implemented the feature `re_eval` that works like the semicolon in Prolog. Finally, to get all the possible answers of a question, it was implemented the feature `all_answers`, that returns an `ARRAY[ANSWER]` which would be empty in case there were no answers.

The operations upon the Knowledge Base are given by the features: `expand`, `contract` and `consolidate.` The first two, `expand` and `contract`, receive a parameter of type `RULE` and permit to remove or to add it dynamically to the Knowledge Base. The last one, `consolidate`, is the responsible for adding the private knowledge of the instance to the class's Knowledge Base, so that any new generated instance have the additional knowledge. Two features of general purpose are provided, `answer_true` and `answer_false`, both `once`, that is to say that the routine only will be executed once. Other invocations will return the same result that the first one, achieving to have an alone instance of each one with the value placed in true or false respectively.

The class `KNOWLEDGE_BASE` is implemented like an array of rules and handles its persistence by means of the class `PERSISTENT_ARRAY`. Each class will store in this array their own rules, but at runtime, the array will be extended to incorporate the ancestor rules, creating a greater knowledge base. For doing this, they will be added together with the own instances rules representing the private knowledge of each object and that has

only existence at runtime. The only way that an object can make its knowledge persistent is adding it to the base class.

The main support classes to MOBI are: `PREDICATE`, `RULE`, `ANSWER`, `SUBSTITUTION` and `ARGUMENT`. The class `PREDICATE` implements the concept of predicate and provides the mechanism to transform a string with the adequate syntax to an instance of it. The class `RULE` represents a Horn clause. The class `ARGUMENT` represents the predicate's arguments. The arguments can be variables, constant string or numerical.

The knowledge base will be created by a set of rules written like Horn clauses, so they will have at most one positive literal. The syntax is similar to Prolog: capital letters represent variables, numbers and strings are constant.

In order to build the Knowledge Base for a class it is necessary to have a rule editor to write the rules, validate its syntax and save them to disk. Therefore a tool named RuleTool was created. This tool allows to chose the name of the class for which it is desired to create or modify the Knowledge Base; after this it is possible to add, modify or eliminate rules, and finally gives the option to save the knowledge to disk. Fig. 4 shows the main screen of RuleTool.

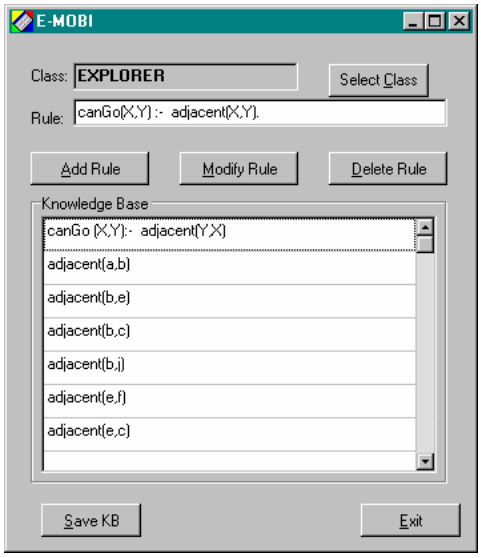


Fig.4 RuleTool

3 CASE STUDY

The case study represents the problem of going from a place to another. For this, a class `EXPLORER` inheriting from MOBI is defined, in such way that its instances can be moved for different positions in the (virtual) world, to arrive at the asked place to go. In this case *the explorer can perceive*. Being in a position it is possible to realize if the nearby

positions are or not *passable*. This information is not in the knowledge base, since to know if a position is passable the explorer must arrive at a contiguous position and watch (to perceive). The perception can be given by external sensors. In the sample we simulated it with one feature *perceive* that contains the perceptions in each position in order to obtain a realistic simulation.

Our explorer learns, by means of his experience, the paths he would not have to take again if he wants to arrive to the same objective. During the exploration made searching the destination, he is learning which positions are impassable and which do not lead to the destination wished from the source (or possibly from some of the positions already taken).

The implementation sample for this case tries to go through the world outlined in the graph of figure 5, where the nodes are the possible positions or states and the arcs indicate the path among them. The darkest nodes indicate the impassable positions (that the explorer does not know until arriving at a contiguous place).

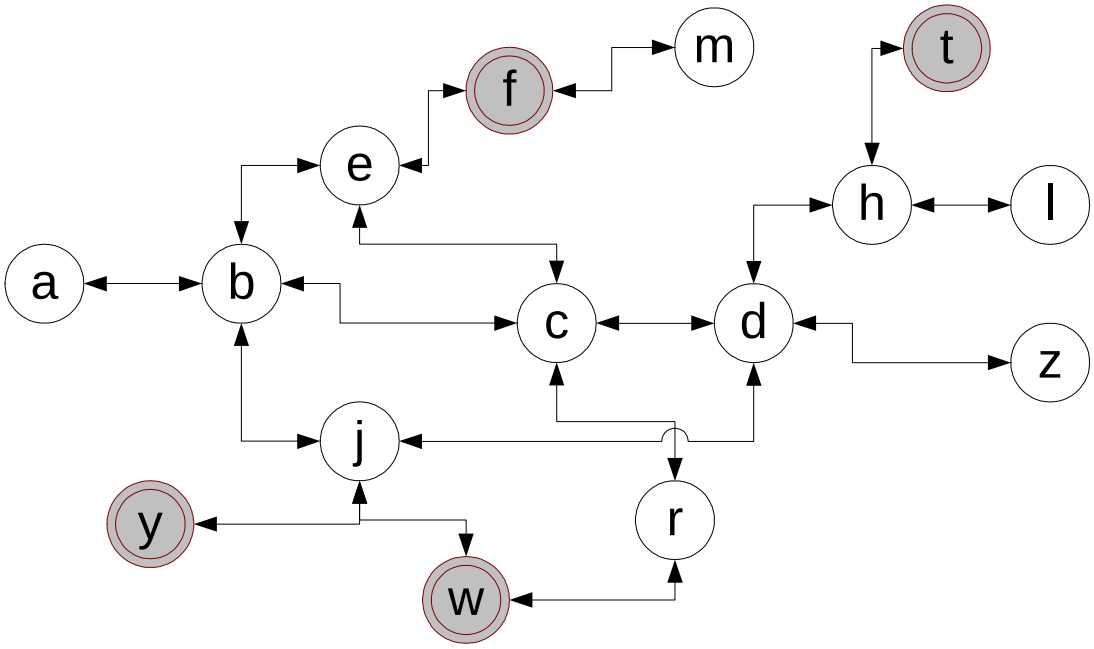


Fig.5 World to Explorer

According to the graph, knowledge bases of the explorer would be:

```
canGo(X,Y) :-  adjacent(X,Y).
canGo (X,Y):-  adjacent(Y,X).
adjacent(a,b).
adjacent(b,e).
```

```
adjacent(b,c).
adjacent(b,j).
adjacent(e,f).
adjacent(e,c).
adjacent(j,c).
adjacent(j,d).
adjacent(c,d).
adjacent(d,h).
adjacent(h,t).
adjacent(d,z).
adjacent(h,l).
adjacent(f,m).
adjacent(j,y).
adjacent(j,w).
adjacent(r,w).
adjacent(c,r).
```

The contract form of the EXPLORER class is the following:

```
class interface EXPLORER

inherit   MOBI   end

creation
   make

feature

   make

   go (source, destiny: STRING): PATH
       -- the agent tries to arrive at the destiny from the
       -- source and return a path between source and destiny.

   steps: INTEGER

   perceive (position: STRING): ARRAY [STRING]
--it simulates an outer perception and it returns an Array of
--impassable places. If all places are passable, then the
--Array is empty

   next_state (path_act: PATH): STRING
      require
         there_is_element: (path_act.length) > (0)

   can_go (pos: STRING): BOOLEAN
```

```
end interface -- class EXPLORER
```

It is possible to appreciate that the interface of the explorer is very simple, basically indicates `go(source, destiny)` and the explorer is responsible to made it; `perceive` and `next_state` are explained by themselves. The first is the simulation of the perception and the second one is responsible to give back the next valid position from the actual place, discarding impassable (perceived as not passable) and closed ways. Thus, the explorer learns from his experience, in the future the amount of steps to go to the same destiny is going to be smaller. If he consolidates his knowledge other agents will not have to prove impassable ways.

A call to an explorer is of the following form:

```
!!explorer.make.
 path := explorer.go( "a","z").
```

He produces the following path: *"a"-> "b"->"e" ->"c" ->"d" ->"z"* with a total of **10** (ten) steps. If the same route is requested to him again, it will give back the same way but with only 6(six) steps, since he learned not to turn aside himself by ways without exit. In addition, if now he is requested to go from **b** to **z**, it takes **5** (five) steps. The private knowledge base (of the instance) after the first invocation is:

```
impassable (f).
impassable (t).
noWay(l,a,z).
noWay (l,b,z).
noWay(l,e,z).
noWay(l,c,z).
noWay(l,d,z).
noWay(l,h,z).
noWay(h,a,z).
noWay(h,b,z).
noWay(h,c,z).
noWay(h,e,z).
noWay(h,d,z).
```

Before each movement, the explorer perceives to incorporate the impassable places in the knowledge base; if he enters in a way without exit he goes back and adds the knowledge of the nodes because they do not lead to the destiny. In each movement finds out the adjacent positions and as soon as is asked to go to a node, it consults the knowledge base to see if it is impassable and does not lead to the objective.

The tests made with the simulator show that the first time to cross until z it makes ten visits whereas for the second request it makes only six visits. Explorer learned, and during the second route it took fewer steps to arrive at a known destination. In addition if

the knowledge is consolidated, any new explorer trying to cross the same single way will make six visits.

4 CONCLUSIONS

The implementation made shows the intelligent development of agents using a multiparadigm model. In [Rus 96] agents are defined of the following form: *"Agents are entities who receive stimulus of environment through sensors and act in this environment starting off of the stimulus"*. In the presented case study the explorer instances act like intelligent agents.

Agents can be implemented interacting with others or in a same space of directions or network. Each agent maintains his knowledge base in a local form, but could make inquiries to others agents located in remote equipments. It is possible that several instances of a class interact, each one with different remote agents to consolidate the common knowledge. Some of the future lines of work include the creation of a framework for distributed processing in Internet using the advantages of E-mobi; and in addition the extension of the evaluation algorithm to support diverse strategies and the incorporation of uncertain knowledge.

REFERENCES

[Gam 95] Erich Gamma, Richard Hel, Ralph Johnson, John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison Wesley. 1995.

[Jez 99] Jean-Marc Jézéquel, Michel Train, Christine Mingins. *Design Patterns and Contracts*, Addison-Wesley, 1999.

[Llo 87] Lloyd J.W., *Foundations of Logic Programming*, Second Edition – N. York 1987

[Mey 92] Bertrand Meyer, *Eiffel the language*, Prentice Hall, 1992

[Mey 97] Bertrand Meyer, *Object-Oriented Software Construction* 2nd Edition, Prentice Hall, 1997

[Mit 02] Mitchell Richard, McKim Jim, *Design by Contract, by Example*, Addison-Wesley, 2002

[Man 01] Andrea Manna, Gerardo Rossel "MOBI Modelo de Objetos Inteligentes". Licenciate Thesis. Faculty of Exact and Natural Sciences. University of Buenos Aires, 2001.

[Rob 65] Robinson J.A. “A Machine Oriented Logic Based on the Resolution Principle”, *Journal of the ACM*, 12, 1965.

[Rus 96] Stuart Russell, Peter Norving. *Artificial intelligence a modern approach*, Prentice Hall - 1996

[Ung 87] David Ungar and Randall B. Smith. “Self: The Power of Simplicity”, *Proceedings OOPSLA ‘87*

About the authors

Gerardo Rossel is professor at Open University Interamericana and National University of Lanus, Argentina. He is researching in object-oriented technology and web engineering for Ph.D thesis at Department of Computer Science, University of Buenos Aires, Argentina. He is also Chief Scientist at UpperSoft (http://www.uppersoft.com.ar). He can be contacted at grossel@dc.uba.ar.

Andrea Manna is attending professor at the Department of Computer Science of the University of Buenos Aires and adjunct professor at the National University of Lanus, Argentina. She is also Chief Software Architect at UpperSoft which provides consulting, training and development in object technology and artificial intelligence. She is working in object technology since 1993. She can be contacted at amanna@dc.uba.ar.