OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods

Oscar Pastor, Emilio Insfrán, Vicente Pelechano, José Romero, José Merseguer
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València
Camí de Vera s/n
46071 Valencia (Spain)
{opastor|einsfran|pele|jmerse|jromero}@dsic.upv.es

Abstract

OO-Method is an OO Methodology that blends the use of formal specification systems with conventional OO methodologies based on practice. In contrast to other approaches in this field ([Jun95,Esd93]), a set of graphical models provided by the methodology allows analysts to introduce the relevant system information to obtain the conceptual model through a requirements collection phase, so that an OO formal specification in Oasis ([Pas92, Pas95-1]), can be generated at any time. This formal specification acts as a high-level system repository. Furthermore, a software prototype which is functionally equivalent to the Oasis specification is also generated in an automated way. This is achieved by defining an execution model which gives the pattern for obtaining a concrete implementation in a declarative or an imperative software development environment (depending on the user choice). The methodology is supported by a CASE workbench.

1. Introduction

In the context of the object paradigm, several OO methodologies have emerged to deal with the set of OO methods to be used to model and correctly implement an information system. Two main approaches can be distinguished:

- what could be called *conventional* OO methodologies, that come from practical use in industrial software production environments, which do not have a formal basis and which often use classical structured concepts together with the introduction of OO features ([Wir90],[Rum91],[Jac92], [Boo94],[Col94]). Recent proposals are trying to create a unified framework for dealing with all the existing methods (UML [BRJ96]), with the implicit danger of providing users with an excessive set of methods that have an overlapping semantics.
- use of OO *formal specification languages* (Oblog [Ser87,Esd93], Troll [Jun91,Har94], Albert [Dub94], Oasis), which have a solid mathematical background and deducible formal properties such as terms of soundness and completeness.

Our contribution to this state of the art is based on the idea that these two approaches can be mixed. This mixing offers some advantages: the use of such OO formal languages can help designers to detect and eliminate ambiguities and elements of dubious utility. The use of conventional OO methodologies permits us to take advantage of the accumulative experience coming from the industrial context. The research work developed at the DSIC-UPV has been directed towards designing and implementing an OO software production environment that aims to combine the pragmatic aspects attached to the so called conventional methods, with the good formal properties of the OO specification languages.

In contrast to other works in this area ([Wie93,Kus95]), our approach is to use this combination of approaches in a graphic, OO conceptual modeling environment which collects the system properties considered relevant for building a formal, textual OO specification in an automated way. This formal OO specification constitutes a high-level system repository. Furthermore, the definition of a concise execution model and the mapping between the specification language and the execution model notions, makes it possible to build an operational implementation of a software production environment allowing for real automated prototyping, by generating a complete system prototype (including statics and dynamics) in the target software development environment. A CASE workbench which supports this working environment in a unified way is currently available for prototyping purposes.

This blend has produced the OO-Method methodology presented in this paper and is based on OASIS as a formal OO specification language. Our intention is to give a clear description of the most relevant features of the approach, introducing the basic ideas on OO conceptual modeling that are in the basis of the work in section 2, and explaining the main OO-Method features as a methodological approach in section 3. The methods used to capture the system properties in order to produce what we will call a conceptual model will be shown. Subsequently we will show how to represent this model in a particular software development environment according to an abstract execution model, which will fix the operational steps to follow when we want to give a concrete system implementation. A software prototype which is functionally equivalent to a system specification can be obtained in the context of the methodology. We will describe the code generation strategy used. Finally, a view of the CASE tool that has been built to support the methodology will also be introduced.

2. The OO-Method Approach

Nowadays, it is considered mandatory for an OO methodology to cover the following aspects:

- Classes and objects
- Abstraction
- Encapsulation
- Inheritance and Aggregation to deal with complex classes
- Interobjectual Communication

However, the current proposals share a common weakness: the value of the conceptual modeling efforts when the development step is reached is unclear, mainly because it is not possible to produce an accurate code which is functionally equivalent to the system requirements specification. We should be able to produce code in an interactive way from the very beginning of the requirements specification step, and not generate only static templates for the component system classes as most OO CASE tools already do. We should be able to generate a complete programming environment including statics and dynamics. This kind of functional rapid prototyping would allow analysts to show the users a comprehensive image of the application state at any given moment, making it possible to detect analysis errors or misunderstandings as soon as they are originated. Furthermore, system designers

would have a validated starting point for their development tasks, avoiding having to start from scratch.

If we work in a declarative environment, the programs generated are theories of a given logic where the three concepts of machine computation, logic deduction and satisfaction in a theory's standard model are equivalent. In this case, a final software product which is *formally equivalent* to the system specification can be obtained using declarative programming languages with a well-defined declarative and operational semantics and with equivalent results between them.

If the target environment is imperative, we lose the quoted declarative properties. However, we can generate a prototype which is *functionally equivalent* to the requirements specification, if we clearly define a mapping between the conceptual and the execution model. This automated prototyping policy (introduced as a code generation strategy later on in this paper) constitutes an important improvement with respect to the current state of the art of the field.

In summary, all these ideas lead us to the OO-Method proposal. OO-Method is an OO methodology, which is intended to overcome these problems and whose contribution is based on the following basic principles:

1. to give support to the OO conceptual modeling notions,
2. to join OO formal method concepts with practical and widely used OO methodologies,
3. to provide an automated prototyping environment, including complete code generation (data and behaviour) in both declarative and imperative programming environments.

3. The Methodology

OO-Method is an Object-Oriented Software Production Methodology whose phases are shown in Figure 1. Basically, we can distinguish two components: the conceptual model and the execution model.

When facing the conceptual modeling step of a given Information System, we have to determine the components of the object society without being worried about any implementation considerations. The problem at this level is to obtain a precise system definition, and this is the conceptual model.

Once we have an appropriate system description, a well-defined execution model will fix the characteristics of the final software product, in terms of user interface, access control, service activation, etc., in short, all the implementation-dependent properties.

In this context, we start with an Analysis step where three models are generated: the Object Model, the Dynamic Model and the Functional Model. They describe the Object Society from three complementary points of view within a well-defined OO framework. For these models we have preserved the names used in many other well-known and widely-used OO methodologies, even if the similarities are purely syntactic as can be seen throughout this paper.

From these analysis models, a corresponding formal and OO Oasis specification (the OO-Method design tool) can be obtained in an automated way.

This is done through an automatic translation process. The resultant Oasis specification acts as a complete system repository, where all the relevant properties of the component classes are included.

According to the execution model, a prototype which is functionally equivalent to the specification is built in an automated way. This may be done in both declarative (Prolog-based) [Can95] and imperative environments (specially those visual OO programming environments that are widely used nowadays). The code generation strategy is independent of any concrete target development environment, even if at the moment our selected environment for automated code generation are Visual C++, Delphi, Java, Visual Basic and PowerBuilder.

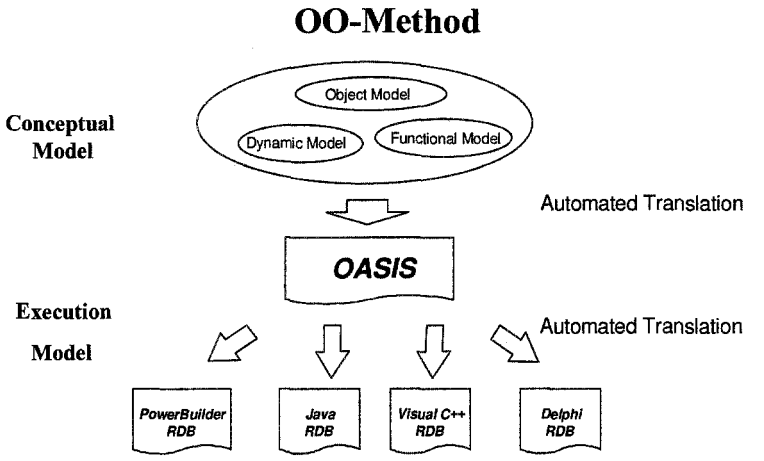


Fig. 1. Phases of OO-Method.

Next, we explain the characteristics of the three models (object, dynamic and functional) that constitute the **conceptual model**, introduce the **execution model** features and explain the conversion strategy from the former to the latter.

3.1Conceptual model

Object Model

The Object Model is represented by means of a Class Configuration Diagram (CCD), a graphic model where system classes are declared, including their attributes and services. Aggregation and inheritance hierarchies are also graphically depicted representing class relationships. Additionally, agents are introduced to specify who can activate each class service. Classes are the basic modeling units. A class is represented by a rectangle with three areas:

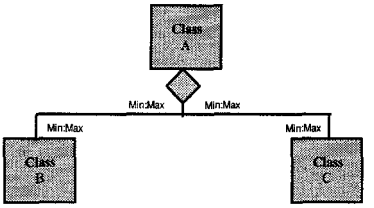
- a header with the class name.
- a static component where attributes are declared.
- a dynamic component where services are introduced, distinguishing among new and destroy events, and among private and shared events.

Shared events are connected by solid lines in the CCD. Client classes (agents) of a given service are represented by dotted lines joining every potential

client class with the corresponding server class, capturing the client system view in an easy and intuitive way.

OO-Method deals with complexity by introducing aggregation and inheritance hierarchies.

We represent the aggregation relationship between two classes including its cardinality (minimum and maximum) to determine how many components can be attached to a given container and how many containers a component class can be associated with. See Figure 2.



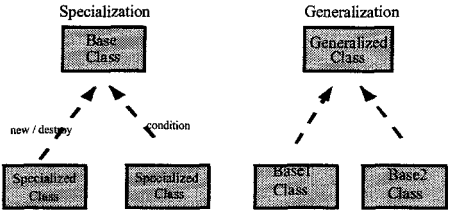


Fig. 2. Aggregation relationship.
Fig. 3. Inheritance relationship.

Inheritance is graphically depicted as an arrow from a given subclass to its superclass. This arrow can be labeled with a condition of specialization, or with the events that activate/cancel the child role, respectively. See Figure 3[1].

Next, the CCD corresponding to a classical Library Information System is shown in the Figure 4. As a basic explanation (for reasons of brevity), we assume that as usual in such a System, there are *readers*, *books* and *loans* relating a book to the reader who orders it. Readers can 'play the role' of *unreliable readers*, if their return dates expire. *Librarian* and *reader* instances are declared as active objects.

---

[1] *This is how inheritance is dealt with in Oasis, distinguishing between permanent and temporal specialization. The permanent case refers to child instances created when the ancestor instance is created, and they need a condition which is built on constant attributes. Temporal specialization (role) appears when a superclass event happens or a condition built on variable attributes holds.*

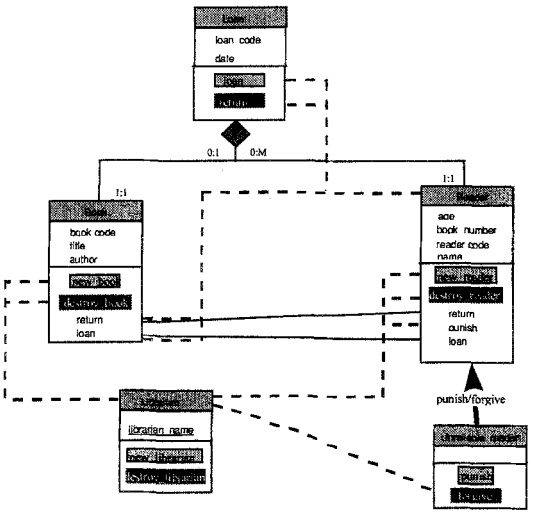


Fig. 4. CCD that represents the Object Model of the Library Information System.

Dynamic Model

The Dynamic Model is used to specify valid object lives and interobjectual interaction. To describe valid object lives, we use State Transition Diagrams (STDs, one for each class). To deal with object interaction, we introduce an Object Interaction Diagram (OID), one for the whole System.

State Transition Diagram

STDs are used to describe correct behaviour by establishing valid object lives. By valid life, we mean a right sequence of states that characterizes the correct behaviour of the objects for every class. In this context states denote the different available situations for class objects, and are depicted using a circle labeled with the state name.

When an object does not exist, a blank circle represents this "state" of non existence, and will be the source of initial transition labeled by the corresponding new event. A bull's eye is used to represent the post-mortem state.

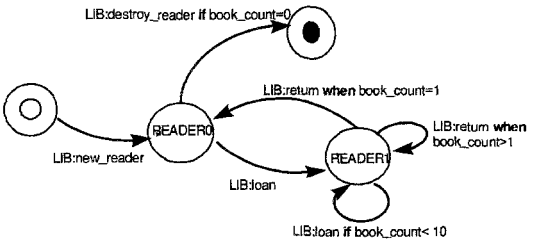


Fig.5 STD for a READER.

Transitions represent valid changes of state that can be constrained by introducing conditions. They follow the syntax shown below:

event | action | transaction [if precondition] [when control condition]

where precondition is a condition defined on the object attributes that must hold for a service to occur and a control condition is a condition that avoids the possible non-determinism for a given action. An example of STD can be seen in Figure 5.

Object Interaction Diagram

The object interactions are represented by diagrams of this kind. We declare two basic interactions:

- *triggers*, which are services of objects which are activated in an automated way when a condition is satisfied by an object of the same or another class.
- *global interactions*, which are transactions involving services of different objects. With these global interactions, interobjectual transactions can be declared. Formally, they can be seen as a local service of the aggregation among the classes providing the services that constitute the global interaction.

Basically, we represent classes in the OID as boxes with a header including the class name. Class services are declared as smaller boxes inside the corresponding class box. The class service boxes are connected when one of the previous types of interactions is defined. Triggers are introduced by starting the corresponding solid line in the header of the class and ending it in the triggered action, and global interactions are introduced by connecting the involved services with a common global interaction identifier (GIid). The general model for an OID can be seen in Figure 6 and 7.

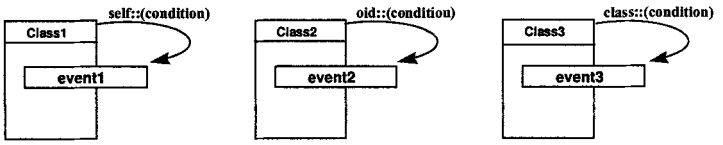


Fig. 6. Trigger Relationships

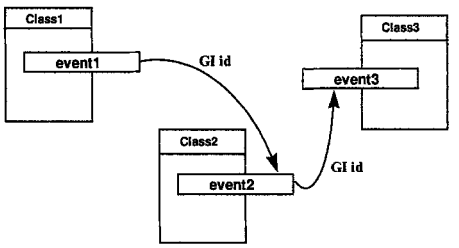


Fig. 7. Global Interaction

Functional Model

After declaring object attributes and services in the Object Model and valid life cycles and object interactions in the Dynamic Model, the aim of the Functional Model is to capture semantics attached to any change of state in an easy and an

intuitive way. This model specifies the effect of an event on its relevant attributes through an interactive dialogue. The value of every attribute is modified depending on the action that has been activated, the involved event arguments and the current object state.

The specification of an action effect should be made declaratively, as proposed in Oasis. However, a good specification requires a solid formal basis for any analyst. To solve this situation, the OO-Method provides a model where the Analyst only has to categorize every attribute among a predefined set of three categories and introduce the relevant information depending on the corresponding selected category.

This classification of attributes [Pas96-2] is a contribution of this method and gives a clear and simple strategy for dealing with the task of generating the Execution Model. At the same time, it opens the door to being able to include this information in an Oasis specification in an automated way.

There are three types of attributes: *push-pop*, *state-independent* and *discrete-domain based* attributes.

Push-pop attributes are those whose relevant events increase or decrease their value by a given quantity. Events that reset the attribute to a given value can also exist.

An example of this category is the *book_number* of the *reader* class, with *REA:loan* as increasing action and *REA:return* as decreasing one (*REA* is a variable of type *reader*).

Attribute : book_number *Category* : push-pop

Action Type	Action	Effect	Evaluation Condition
Incr.	REA:loan	+1	
Decr.	REA:return	-1	

Fig. 8. Push-pop attribute *book_number* of the *reader* class.

State-independent attributes have a value that depends only on the latest action that has occurred. Once a relevant action is activated, the new attribute value of the object involved is independent of the previous one. In such a case, we consider that the attribute remains in a given state, having a certain value for the corresponding attribute. We can introduce the attribute *bookshelf* of the *book* class as an example. A *book* has a *bookshelf* assigned when the event *locate*(B) is activated. When this event occurs, *bookshelf* takes the argument value independently of any previous value.

Attribute : bookshelf *Category* : state-independent

Carrier Action	Action Effect	Evaluation Condition
LIB:locate(B)	=B	

Fig. 9. State-independent attribute *bookshelf* of the *book* class.

Discrete-domain valued attributes take their values from a limited domain. The different values of this domain model the valid situations that are possible for objects of the class. Through the activation of carrier actions (that assign a given domain value to the attribute) the object reaches a specific situation. The object abandons this situation when another event occurs (a "liberator" event). As an example, let's consider the *available* attribute of the *book* class. The available value tells us what the current book situation is. The carrier event (*loan*) lets the object into

a situation where available has the value false. The situation is abandoned when the event *return* is activated.

Attribute : available *Category* : discrete-domain valued

Actual Value	Action	New Value	Evaluation Condition
TRUE	REA:loan	FALSE	
FALSE	REA:return	TRUE	

Fig. 10. Discrete-valued attribute *available* in the *book* class.

All this information, which constitutes the system description, has a textual representation in Oasis. The specification is obtained at any moment by executing an automated process of translation that converts the collected graphic information into a textual OO specification that constitutes a complete, formal System Repository.

3.2 Execution Model

Once all the relevant system information in the specification that we have called conceptual model is collected , the execution model has to accurately state the implementation-dependent features associated to the selected object society machine representation. More precisely, we have to explain the pattern to be used to implement object properties in any target software development environment.

Our idea at this point is to give an abstract view of an execution model that will set the programming pattern to follow when dealing with the problem of implementing the conceptual model. This execution model has three main steps:

1. *access control:* first, as users are also objects, the object logging in the system has to be identified as a member of the corresponding object society.
2. *object system view:* once the user is connected, he must have a clear representation of which classes he can access. In other words, his object society view must be clearly stated, precising the set of object attributes and services he will be allowed to see or activate, respectively.
3. *service activation:* finally, after being connected and having a clear object system view, the object will be able to activate any available service in the user's world view. Among these services, we will have event or transaction activation served by other objects, or system observations (object queries).

Any service execution is characterized as the following sequence of actions:

1. *object identification*: as a first step, the object acting as server has to be identified. This object existence is an implicit condition for executing any service, except if we are dealing with a *new*[2] event. At this moment, their values (those that characterize its current state) are retrieved.
2. *introduction of event arguments*: the rest of the arguments of the event being activated must be introduced.
3. *state transition correctness*: we have to verify in the STD that a valid state transition exists for the selected service in the current object state.

---

[2] *Formally, a new event is a service of a metaobject representing the class, which acts as object factory for creating individual class instances. This metaobject (one for every class) has as main properties the class population attribute, the next oid and the quoted new event.*

4. *precondition satisfaction*: the precondition associated to the service that is going to be executed must hold. If not, an exception will arise, informing that the service cannot be activated because its precondition has been violated.
5. *valuation fulfilment*: once the precondition has been verified, the induced event modifications are effective in the selected persistent object system.
6. *integrity constraint checking in the new state*: to assure that the service activation leads the object to a valid state, we must verify that the (static and dynamic) integrity constraints hold in this final resulting state.
7. *trigger relationships test*: after a valid change of state, and as a final action, the set of rules condition-action that represent the internal system activity have to be verified. If any of them holds, the corresponding service activation will be triggered. It is the analyst's responsibility to assure the termination and confluence of such triggers.

The previous steps guide the implementation of any program to assure the functional equivalence among the object system description collected in the conceptual model and its reification in a software programming environment according to the execution model.

Next, we are going to present the code generation strategy used in the implementation of the previous execution model in a well-known Windows95 environment, which opens up the possibility of creating a CASE tool that, starting from a set of graphical OO models obtained during the conceptual modeling step (according to OO-Method) can generate a functional software prototype at any time.

3.3 Code generation strategy

Once an abstract execution model has been introduced, we will have different concrete implementations of this execution model for different software development environments. In this paper, we focus on the implementation of the execution model in a Windows95 context, but it must be noted that other concrete and alternative implementations are currently being been developed emphasizing one using Java in an intranet environment. It is important to note that the representation of the conceptual model in the selected execution model is done according to the principles introduced above, thus generating a prototype in an automated way by adapting the code generation strategy that we present to the particularities of the target development environment.

The execution model implementation selected for a Windows95 environment keeps in mind the main principles attached to such a environment. Basically, this means that we have:

- to reproduce the user's mental image of the system, within an OO world view. Users generally expect an application to operate in accordance with its nature, and the OO paradigm provides an operational framework to properly represent a system as a society of interacting objects, where every individual object can access other system component objects and can activate those services it is allowed to. To ensure this consistency, the interfaces built have to resemble the user's environment. They also have to be consistent, complying with the standards in presentation (what the user sees), behaviour (how the application

reacts), sequencing (how the dialogs are sequenced) and functionalities (how actions are carried out). Finally they have to be transparent, meaning that the purely technical application mechanisms must be completely transparent to the user.

- to give control to the user. It is the user who must control the application and not the contrary.

To properly implement the set of system classes in a standard Windows95 software development environment, we have to deal with a static and a dynamic point of view. The static one will fix the relational database schema corresponding to the system specification. This automated relational generation is out of the scope of this presentation and is explained in depth in [Pas95-2]. In short, every class is converted into a relation, having the attribute information included in the class specification. Aggregation and inheritance are treated by defining the corresponding foreign keys according to the collected complex class properties. Next, we are going to focus on dynamics explaining the appearence of the prototype which is automatically generated.

The code generation process creates four types of windows as we can see in Figure 11:

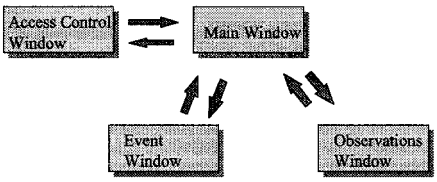


Fig.11 Overview of the generated code structure.

- *Access Control Window*: this is the log-in window, where the corresponding active user has to be identified. This is done by introducing its object identifier, class name and password. The identification is verified on the database to ensure that the object exists. Once the object is incorporated to the system, it will see the available system class services through menu items of the main menu.

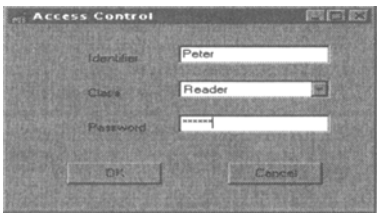


Fig.12 Access Control Window

- *Main Window*; it characterizes the system view that the connected object has. All the services of the classes are requested through it. It has the following options:

  ◊ the typical File item option of Windows applications.

◊ for every class, a pull-down menu including an item for observations (queries), a section with its descendent classes (if any) and a last section with the available class services.
◊ an interactions item, which allows for the activation of global interactions.

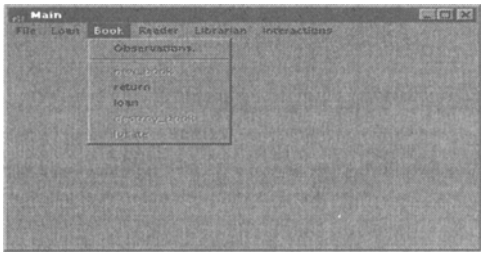


Fig.13 Main Window.

- *Event window*, where the corresponding arguments are introduced and the induced actions are executed through the OK control button.
- *Observations window*; this screen is intended to be a Query By Example pattern where the user can see the results of any query done over the current object state.

Finally, we will give a quick look at the OO-Method CASE tool.

4. The OO-Method CASE Tool

The OO-Method CASE Tool [Pas96-1] provides an operational environment that supports all the methodological aspects of OO-Method. It simplifies the analysis, design and implementation of Information Systems from an object-oriented perspective, providing a comfortable and friendly interface for elaborating the OO-Method models taking advantages of Windows95. The CASE Tool is being used at this moment in the resolution of real complex systems, in the context of a R&D project carried out jointly by the Valencia University of Technology and Consoft S.A.

The most interesting contribution of this CASE environment is its ability to generate code in well-known industrial software development environments from the system specification, what constitutes an operational approach of the ideas of the automated programming paradigm: analysts collect information, and can generate a formal OO system specification, and a complete (including statics and dynamics) software prototype which is functionally equivalent to the quoted system specification whenever the analysts want.

When the CASE Tool is executed, we are placed on a blank blackboard that represents the CCD where we can draw classes and their properties. By selecting one of the classes on the CCD the user can change to the STD dynamic model. The OID completes the dynamic model. In addition to these static and dynamic points of view the user has to fill the functional model information through friendly and interactive dialogs.

The Figure 14 shows a picture of the CASE Tool. The main menu of the tool has the typical items of an editing tool and also allows the user to enter in textual mode the OO-Method models. Two remarkable items are the Project item that includes the options for the Analysis (object, dynamic and functional models), Design (Oasis code, generated in an automated way) and Implementation (Visual C++, Delphi,... code) steps, and the View item which allows the user to manage the complexity of the graphic diagrams.

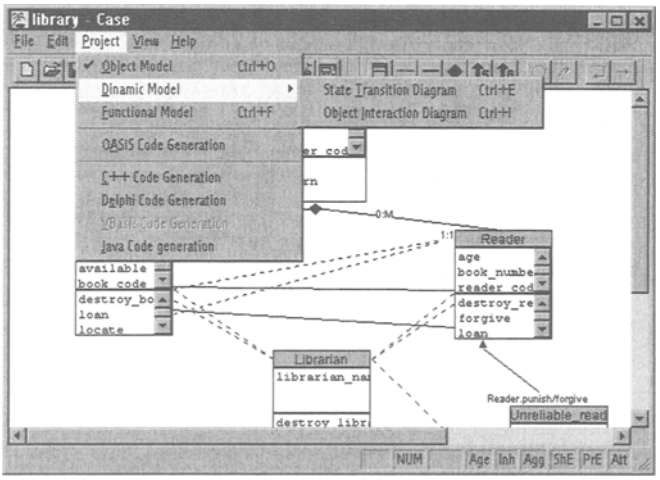


Fig. 14 OO-Method CASE Tool

5. Conclusions

The main aspects of the presented work are the following:

1. A complete OO methodology for dealing with all the Software Production Process phases has been introduced. This methodology uses a formal OO specification language (Oasis) as a central, well-defined repository, from which executable application prototypes can be obtained at any given moment.
2. A CASE tool for Rapid Prototyping is provided. It is embedded in the methodological OO context of OO-Method, having as basic property that the collection of system requirements generates a prototype to be run by final users in order to validate this process of requirements engineering.
3. On the basis of our approach, we find an operational environment blending classical, widely-used OO methods with formal specification languages, complementing their different backgrounds: software development practice on the one hand, and a mathematical theory background on the other hand.

References

[Boo94] Booch,G. *OO Analysis and Design with Applications*. Addison-Wesley, 1994.

[BRJ96] Booch,G.,Rumbaugh,J.,Jacobson,I. *Unified Modeling Language. Version 0.91*. Rational Software Corporation.

[Can95] Canós,J.H.;Penadés,M.C.Ramos,I. *A Knowledge-Based Arquitecture for Object Societies.* Proc. of DEXA-95 (Workshop), pags: 18-25, London, 1995

[Col94] Coleman,D.;Arnold,P.;Bodoff,S.;Dollin,S.;Gilchrist,H.;Hayes,F.;Jeremes,P. *Object-Oriented Development; The Fusion Method.* Prentice-Hall 1994

[Dub94] Dubois,E.;Du Bois,Ph.;Petit,M.;Wu,S. *ALBERT:A Formal Agent-Oriented Requirements Language for Distributed Composite Systems.* In Proc. CAiSE'94 Workshop on Formal Methods for Information System Dynamics, pags: 25-39, University of Twente, Technical Report 1994.

[Esd93] ESDI S.A., Lisboa. OBLOG CASE V1.0- User's Guide

[Har94] Hartmann T.,Saake,G.,Jungclaus,R.,Hartel,P.,Kusch,J. *Revised Version of the Modeling Language Troll (Troll version 2.0).* Technische Universitat Braunschweig, Informatik-Berichte, 94-03 April 1994.

[Jac92] Jacobson I.,Christerson M.,Jonsson P.,Overgaard G. *OO Software Engineering , a Use Case Driven Approach.* Reading, Massachusetts. Addison -Wesley.

[Jun91] Jungclaus, R., Saake, G., Sernadas, C. *Formal Specification of Object Systems.* Eds. S. Abramsky and T. Mibaum Proceedings of the TapSoft´s 91, Brighton. Lncs. 494, Springer Verlag 1991, pags. 60-82.

[Kus95] Kusch,J.; Hartel,P.;Hartmann,T.;Saake,G. *Gaining a Uniform View of Different Integration Aspects in a Prototyping Environment.* Proc of DEXA-95, pags. 35-42, LNCS 978, Springer-Verlag, 1995

[Pas92] Pastor, O.;Hayes,F.;Bear,S. *OASIS:An OO Specification Language.* Proc. of CAiSE-92 Conference, Lncs (593), Springer-Verlag 1992, pags: 348-363.

[Pas95-1] Pastor,O., Ramos, I. *Oasis 2.1.1: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach,* October 95 (3 ed).

[Pas95-2] Pastor,O.;Garcia,R.;Cuevas,J. *Implementation of an OO Design in an Oracle7 Development Environment.* Proc. of the European Oracle Users Group Conference, EOUG-95. Vol.4 pags: 35-47, Firenze (Italy).

[Pas96-1] Pastor,O., Barberá, J.M., Merseguer, J., Romero, J., Insfrán, E.: *The CASE OO-METHOD graphic environment description.* Tech. Report, ITI-DT-96.

[Pas96-2] Pastor,O., Pelechano V., Bonet B., Ramos I. : *An OO Methodological Approach for Making Automated Prototyping Feasible.* Proceedings of DEXA96, Springer-Verlag, September 1996.

[Ser87] Sernadas,A.;Sernadas,C.;Ehrich,H.D. *OO Specification of Databases: An Algebraic Approach.* In P.M.Stocker, W.Kent eds., Proc. of VLDB87, pags: 107-116, Morgan Kauffmann, 1987.

[Rum91] Rumbaugh J.,Blaha M., Permerlani W., Eddy F.,Lorensen W. *Object Oriented Modeling and Design.* Englewood Cliffs, Nj. Prentice-Hall.

[Wir90] Wirfs-Brock R., Wilkerson B., Wiener L., *Designing Object Oriented Software.* Englewood Cliffs, Nj. Prentice-Hall.

[Wie93] Wieringa, R.J., Jungclaus, R., Hartel, P., Hartmann, T., Saake, G., *OMTROLL Object Modeling in TROLL.* Proc. of the International Workshop on Information Systems - Correctness and Reusability (IS-CORE'93). Hannover, September 1993. Udo W. Lipeck, G.Koschorrek (eds.).