From Object-Oriented Conceptual Modeling to Component-Based Development

para enviar a DEXA'99

2nd March 1999

Abstract

Conventional OO methodologies have to provide a well-defined Component-based development (CBD) process by which the community of software engineers could properly derive executable software components from requirements in a systematic way. The move toward CBD requires existing OO conceptual modeling approaches to be reconsidered. In this paper we present a proposal to support CBD in an OO Method based on a formal OO model. The key element of this proposal resides on the concept of execution model. The execution model defines a model and an architecture that provides a pattern to generate software components from OO conceptual models.

Conceptual modeling patterns have a corresponding software representation in the quoted component-based architecture. The implementation of these mappings from problem space concepts to solution space representations opens the door to the generation of executable software components in an automated way.

1 Introduction

Much of the existing work in component-based software technology has concentrated on developing infrastructure capabilities and middleware solutions [9, 3, 7] for connecting independent pieces of system functionality. In this context, where new technologies are continuously emerging, application developers require new additional capabilities that include:

- methods for designing CBD solutions that help the organization focus on the major functional pieces of their domain, and how those pieces will interact.
- tools that support specification of business components using techniques that allow the functionality to be described independently of a particular implementation technology.

In the field of OO methodologies [20, 16, 10, 1, 6, 2], the move toward CBD requires existing approaches to be reconsidered. In particular, any method supporting CBD is required to exhibit at least the following 4 key principles [4]:

- a clear separation of component specification from its design and implementation.
- an interface-focused design approach.
- more formally recorded component semantics.
- a rigorously recorded refinement process.

Our contribution to this state of the art is the OO-Method approach [?, 12, 13], that basically is built on a formal object-oriented model (OASIS) and whose main feature is that developers's efforts are focused on the conceptual modeling step, where system requirements are captured according with a predefined, finite set of what we call conceptual modeling patterns because they are a representation of relevant concepts at the problem space level. The full OO implementation is obtained in an automated way following an execution model (including structure and behaviour). This execution model establishes the corresponding mappings between conceptual model constructs and their representation in a particular software development environment.

The main contribution of this paper is the proposal of a component-based architecture for the OO-Method execution model. This proposal has two benefits:

- it provides a pattern for obtaining software components starting from the OO code generated. These components can be dynamically combined to build a software prototype that is functionally equivalent to the specification collected in the conceptual modeling step.
- it provides a framework to execute the specification in the solution space.

This paper is organized as follows: section 2 describes the main OO-Method features to support CBD. Section 3 presents a short description of the diagrams that are used to capture the system properties in order to produce what we call a conceptual model. Subsequently, the underlying OO OASIS formal specification that is obtained when the conceptual modeling step is finished will be introduced. Section 4 describes how to represent this OASIS formal specification in any target software development environment, according to an abstract execution model. The component-based architecture to produce the software components that allow us to link the conceptual model with the abstract execution model in an automated way, is presented in detail. Finally, section 5 presents the conclusions and further works.

2 Towards CBD

The basic idea in the OO-Method proposal to support CBD resides in clearly separate what we call the conceptual model level and the execution model level.

The first one, represents the problem space, and is centered in what the system is. The second one represents the solution space, and is intended to give an implementation in terms of how the system is to be implemented.

There are four key steps that makes the OO-Method approach suitable for component modeling. These steps *are classified in figure 1*:

```
------------------------------------------------------------
PROBLEM SPACE (conceptual model level)
------------------------------------------------------------
1.- Capture a conceptual model.
2.- Obtain a formal specification of the Conceptual Model
    in an automated way.
------------------------------------------------------------
SOLUTION SPACE (execution model level)
------------------------------------------------------------
3.- Generate OO code from the formal specification
4.- Obtain a component specification of the system in an
    automated way.
------------------------------------------------------------
```

Figure 1: key steps in the OO-method approach

The problem space involves the tasks associated to obtain a formal specification of the requirements that must be acomplished by the system. This is a two-step process; capture a conceptual model and obtain an OASIS formal specificacion in an automated way.

This formal specification, acts as a high-level system repository. It is the source for an execution model that must accurately state the implementation-dependent features associated to the selected object society machine representation.

In consequence, the execution model establishes a concrete strategy starting from the OASIS formal specification to generate OO code in the solution space. Having generated the code of the application logic, a component specification of the system can be obtained in an automated way.

The rest of the paper discuss these steps in detail.

3 Problem Space

3.1 Capture a conceptual model

Conceptual modeling in OO-Method collects the system properties using three complementary models:

- The Object Model: a graphical model where system classes including attributes, services and relationships (aggregation and inheritance) are defined. Additionally, agent relationships are introduced to specify who can activate each class service (client/server relationship).

- The Dynamic Model: another graphical model to specify valid object life cycles and interobject interaction. We use two kinds of diagrams:

  1. State Transition Diagrams to describe correct behaviour by establishing valid object life cycles for every class. By valid life, we mean a right sequence of states that characterizes the correct behaviour of the objects.
  2. Object Interaction Diagram to represents interobject interactions. In this diagram we define two basic interactions: triggers, which are object services that are activated in an automated way when a condition is satisfied, and global interactions, which are transactions involving services of different objects.

- The Functional Model: is used to capture semantics attached to any change of an object state as a consequence of an event occurrence. We specify declaratively how every event changes the object state depending on the involved event arguments (if any) and the object's current state. We give a clear and simple strategy for dealing with the introduction of the necessary information. This is a contribution of this method. It allows us to generate a complete OASIS specification in an automated way. More detailed information can be found in [14].

3.2 Obtain a formal specification of the Conceptual Model

Starting from the three previous models, a corresponding formal and object-oriented OASIS specification is obtained in an automatic way. The resultant specification acts as a high-level system repository. This is a main feature in the OO-Method approach: each piece of information introduced in the conceptual modeling step has a corresponding formal counterpart, represented as an OASIS concept. We could view the graphical modeling environment attached to OO-Method as an advanced graphical editor for OASIS specifications.

To have a complete view of the presented approach, we are going to introduce the mappings that, taking as input the graphical information introduced in any OO-Method conceptual model, generates a textual system representation that is a specification in OASIS.

We are going to illustrate this process using an small example. Consider a conventional library system with readers, books, and loans. Figure 2 shows a partial OASIS specification for the class reader.

According to the OASIS class template, let's identify the set of OO-Method conceptual patterns and their corresponding OASIS representation.

From the object model, the system classes are obtained. For each class, we have

- its set of (constant,variable or derived) attributes. In the example, reader_code is defined as constant attribute because their value doesn't change along the entire live of an instance of this class. Book_count is defined as variable attribute, their value represents the number of books that a reader has in a given moment.

```
class reader
constant_attributes
  reader_code : String;
variable_attributes
  book_count : Int;
  private_events
  new_reader() new;
  destroy_reader() destroy;
  punish();
shared_events
  loan() with book;
  return() with book;
constraints
  static book_count < 10;
valuation
  [loan()] book_count= book_count + 1;
  [return()] book_count= book_count - 1;
preconditions
  destroy_reader () if
  book_count = 0 ;
triggers
  Self::punish() if book_count = 10;
process
  reader = new_reader() reader0;
  reader0= destroy_reader() +
           loan() reader1;
  reader1= return() reader0 +
           loan() reader1;
end_class
```

Figure 2: a partial OASIS specification for the class reader

- its set of services, including(private and shared) events and local transactions. Figure 2 shows how new_reader(), destroy_reader() and punish() are defined as private events because they participate in the life of instances of an object class reader. The loan() and return() events are defined as shared events, they participate in the lifes of instances of several object classes. In this case, they belong to lifes of both the reader and book involved instances.

- the integrity constraints specified for the class which state conditions that must be satisfied. In this case, the number of books that a reader has must be lower than 10.

- the derivation expressions corresponding to the derived attributes.

If we are dealing with a complex class (those defined by using the provided aggregation and inheritance class operators), the object model also provide the particular characteristics specified for the corresponding complex aggregated or specialized class.

With the information given by the object model, we basically have the system class framework, where the classes signature is precisely declared. The dynamic model used two kind of diagrams: from the STD, we obtain

- event preconditions (those formula labeling the event transitions) which determine when an event can be activated. In the example, a destroy_reader event can be activated only if the value of the book_count attribute is equal to zero for the involved reader.

- the process definition of a class, where the template for valid object lives are fixed. The execution of processes are represented by terms in a well-defined algebra of processes, which is based on the approach presented in [19]. It allows us to declare possible object lives as terms whose elements are transactions and events.

And from the Interaction Diagram we complete two other features of an OASIS class specification:

- trigger relationships which introduce internal system activity. A triggered action is defined for our example; When the value of the book_count attribute is equal to 10 then the event punish must be activated automatically.
- global transactions (those involving services of different class objects)

And finally, the functional model gives the dynamic formulas related with evaluations, where the effect of events on attributes are specified. If a loan event occurs the book_count is incremented. If a return event occurs the book_count is decremented.

This is how having clearly defined the set of relevant information that can be introduced in a OO-Method conceptual model, the formal specification corresponding to it provides a complete system repository where the system description is completely captured, according to the OASIS object-oriented model. This allows to undertake the implementation process (execution model) from a well-defined starting point, where the involved pieces of information are meaningful because they come from a finite catalog of conceptual modeling patterns, that furthermore have a formal counterpart in OASIS.

The reader can find a more detailed description of OASIS in [11], and the complete description of its semantics and formal foundations in [14].

4 Solution Space

4.1 An abstract Execution Model

The OASIS specification is the source for an execution model that must accurately state the implementation-dependent features associated to the selected object society machine representation. In order to easily implement and animate the specified system, we predefine a way in which users interact with system objects. The template presented in figure 3, is used to achieve this behaviour.

The process starts logging the user into the system (step 1) and providing an object system view (step 2) determining the set of object attributes and services that it can see or activate. After the user is connected and has a clear object system view, the user can activate any available service in their worldview. Among these services, we will have observations (object queries) or local services or transactions served by other objects. Any service activation (step 3) has two steps: build the message and execute it (if possible). In order to build the message the user has to provide information to identify the object server[1] (step 3.1), and subsequently, he must to introduce service arguments (step 3.2) of the

[1] the existence of the object server is an implicit condition for executing any service, unless we are dealing with the service *new*

```
1. Identify the user
2. Obtain the object system view
3. Service activation
    3.1 Identify the object server
    3.2 Introduce service arguments
    3.3 Send the message to object server
    3.4 Check state transition
    3.5 Check preconditions
    3.6 Valuations fulfillment
    3.7 Integrity constraint checking in the new state
    3.8 Trigger relationships test
```

Figure 3: The execution model template process

service being activated (if necessary). Once the message is sent (step 3.3), the service execution is characterized by the occurrence of the following sequence of actions in the server object:

- check state transition (step 3.4) is the process to verify in the OASIS specification that a valid transition exists for the selected service in the current object state.
- the precondition satisfaction (step 3.5) indicates that the precondition associated to the service must hold.

If any of these actions does not hold, an exception will arise and the message is ignored. Otherwise, the process continue with:

- the valuation fulfillment (step 3.6) where the induced service modifications take place in the involved object state.
- to assure that the service execution leads the object to a valid state, the integrity constraints (step 3.7) are verified in the final state. If the constraint does not hold, an exception will arise and the previous change of state is ignored.
- after a valid change of state, the set of condition-action rules that represents the internal system activity is verified. If any of them hold, the specified service will be triggered (step 3.8).

The previous steps guide the implementation of any program to assure the functional equivalence between the object system specification collected in the conceptual model and its reification in an imperative programming environment.

Now we are going to describe how to link the formal specification captured in the conceptual modeling step with the abstract execution model presented above. This process has two steps: first, the proposal of a component-based architecture that provides a framework to execute the specification in the solution space. Second, a concrete strategy to generate the code for the domain classes (application logic), starting from the formal OASIS specification.

4.2 A tiered component architecture for the execution model

The architecture for the execution model is based in the common three-tier architecture [17]. We propose a multi-tiered architecture that includes the separation of responsabilities implied by the classic three-tier architecture. These responsabilities are assigned to software components.

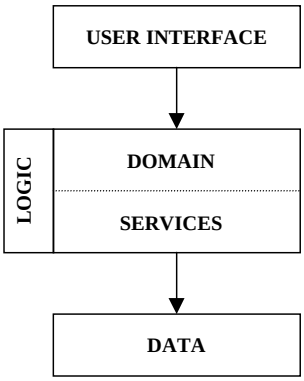

Figure 4: *A n-tiered architecture for the execution model.*

The architecture is shown in figure 4. The user interface layer is composed of components that handle the presentation of information between the user and the application. The application logic is itself composed of the following layers:

- application logic - problem domain: components representing domain concepts that fulfill application requirements according to the OASIS specification.
- application logic - services: non-problem domain components that provide supporting services, such as interfacing with a database (commonly called mediators [18] and/or databasebrokers [5]).

Because the focus of this paper is in the application logic - problem domain layer, we are going to describe in detail the structure of this layer showing:

- how to generate OO code from the OASIS formal specification.
- how to obtain a component specification of the system starting from the code of the domain classes.

4.3 Generating OO code from the formal specification.

In this section, we are going to explain how we translate the formal specification that represents a domain class, to a object-oriented software implementation. The responsibilities of the classes that will be generated are to encapsulate the application logic according to the underlying OASIS specification.

This translation process has two parts. The first one, is about how the domain classes can implement the algorithm to activate a service according to the execution model philosophy. The second one, is about how the OASIS specification is mapped in the code of the domain classes.

4.3.1 Implementing the algorithm to activate services.

The design of the domain classes to implement the algorithm to activate a service, is based upon the Template method pattern [8], see figure 5. In this figure, the most important public method in the OASIS abstract class is activateService(), a template method which takes the name of the service as a parameter and returns a boolean value that indicates if the service has been successfully activated.

```
public abstract class OASIS {
    // template method
    public abstract boolean activateService(String service,
    String objectIdentifier, Hashtable parameters)
    {
        mediator.materialize(objectIdentifier);
        checkPreconditions(service);
        checkStateTransition(service);
        valuations(service,parameters);
        checkIntegrityConstraints();
        checkTriggers();
        mediator.dematerialize(objectIdentifier);
        return true;
    }

    //primitive operations
    protected abstract void checkPreconditions(String service);
    protected abstract void checkStateTransition(String service);
    protected abstract void checkIntegrityConstraints();
    protected abstract void checkTriggers();

}
```

Figure 5: the OASIS abstract class

The activateService() template method defines varying parts of the algorithm. In concrete, the particulars of how the activation of a service can vary depending on the service that is being activated. These responsabilities are left as a primitive operations for subclasses to define, as can be seen in Figure 6.

For instance, checkPreconditions is a primitive operation for the template method activateService(). Consequently, each domain class has to define how to implement the corresponding body of this primitive operation[2]. In this case, the body of this operation must be implemented in the following way: according to

[2]Obviously, we will declare a primitive operation for every checking process in the execution model

```
public class domainClass extends Oasis {
    //implementations of the primitive operations
    protected void checkPreconditions(String service);
    protected void checkStateTransition(String service);
    protected void valuations(String service, Hashtable parameters);
    protected void checkIntegrityConstraints();
    protected void checkTriggers();

    //services of the concrete class
    protected boolean service_1(Hastable parameters);
    protected boolean service_2(Hastable parameters);
    ...
    protected boolean service_n(Hastable parameters);
}
```

Figure 6: the domain classes

the preconditions associated to the domain class in the OASIS specification for that particular service.

Finally, the declaration of the domain class is completed with the services of the class.

Having presented the design pattern that implements the algorithm to activate a service. The next step is to describe the design of the domain classes. Starting from the OASIS specification, we are going to describe how the primitive operations proposed in the algorithm presented above are implemented.

4.3.2 Mapping the OASIS specification into the code of the domain classes.

We are going to introduce these mappings taking into account the example of the library system presented in section 3.2. The programming language used is Java.

Mapping preconditions. Figure 7 illustrates how precondition are mapped into code. From an object-oriented programming point of view, a precondition is a conditional sentence that must be tested. Therefore, preconditions are mapped using a conditional test. If the condition doesn't hold then an exception will arise.

Mapping state transition diagrams. There are basically three different ways for implementing the verification of a valid state transition; conditional logic, the state pattern [8], or an state machine interpreter that runs against a set of state transition rules.

It has been recognized that the applicability of the state pattern may not be suitable if there are many states in the system [8]. The implementation of an state machine interpreter is out the scope of our proposal. Consequently, we

```
protected void checkPreconditions(String serviceName)
throws error {
  if (serviceName.equals("destroy_reader"))
    if (!(book_number == 0))
    throw new error ("Precondition violation.
    The reader has books");
}
```

Figure 7: mapping preconditions for the reader class

have choose the use of conditional logic.

Figure 8 shows, how state diagrams are mapped in the solution space using conditional logic. Particularly, for each concrete state specified in the DTE (in the example of the reader class, these states are nonexistence, reader0, reader1 and post-mortem), we must declare a conditional structure. This structure must ensure that a valid transition exists for the service that is being activated. If this process succeeds, the corresponding change of state is carried out. Otherwise an exception will arise. For example, let's suppose that the current state of a reader object is reader0. Let's suppose that the service that is being activated is the service loan(). The stateTransition method solves that this is a valid transition and set the new state of the reader object to reader1.

```
protected void stateTransition(string service)
throws error {
  if (state.equals("nonexistence"))
    { if (service.equals("new_reader"))
        {state="reader0";
         return;}
    }
  else if (state.equals("reader0"))
    { if (service.equals("loan"))
        {state="reader1";
        return;}
      else if (service.equals("destroy_reader"))
        {state="post-mortem";
        return;}
  else if (state.equals("reader1"))
    { if (service.equals("return"))
        {state="reader0";
        return;}
    }
  throw new error ("State violation. Not exists a valid
  transition")
}
```

Figure 8: mapping the state diagram for the reader class

Mapping valuations. The valuations are mapped in the solution space as a method of the domain class (see valuations method in figure 9). This method is

responsible to check which is the service that must be activated. The execution of the appropiate service implements the change of the object attributes values according to valuations specified in the OASIS specification.

```
protected void valuations(String serviceName,
Hashtable parameters) {
if (serviceName.equals("loan"))
    loan(parameters);
if (serviceName.equals("return")
    return(parameters);
}
```

Figure 9: mapping valuations for the reader class

In the example of the figure 9, we use conditional logic to check which is the service that has been activated. The corresponding method will be activated. This method will modify the attribute values according to the OASIS valuations. Let's suppose that the service loan() has been activated for a reader object. The effect of this service activation in the attribute book_count will be an increment in their value.

Mapping integrity constraints. They are mapped using conditional logic as can be seen in figure 10. In the example, the method constraints will check if the conditional expression book_number < 10 is satisfied.

```
protected void constraints() throws error
{
if (!(book_count < 10))
throws new error ("Constraint Violation.
Borrow limit exceeded";
}
```

Figure 10: mapping constraints for the reader class

Mapping triggers. Traditionally, the effect of a triggering action has been a singular research topic in database community. Specially in the context of active DBMS. A common proposal of a knowledge model for active systems has been the event-condition-action (ECA) rules mechanism [15]. These rules are composed of an event that triggers the rule, a condition that describes a concrete situation, and an action to be performed if the condition is satisfied. In this context, various proposals has been proposed to include these rules mechanism in active systems.

For our purposes, we only consider a subset of ECA rules. We define an OASIS trigger as an active rule where; the event is the current service that is being activated, the condition is the expression that must be evaluated, and the action is the OASIS action that must be activated if the condition is satisfied. We also consider that the effect of the trigger acts in the context of the object society that is being specified.

Taking into account these limitations, we propose a simple strategy of implementation. Triggers are mapped as a method (see method triggers in figure 11), that checks if the condition associated to the trigger (book_count = 10), holds. In this case, the corresponding service (activateService("Punish")), is activated.

```
protected void triggers() throws error
{  if (book_count = 10)
try { activateService("Punish");
} catch (EX_triggers e) {throws e};
}
```

Figure 11: mapping triggers for the reader class

This is how starting from the formal OASIS specification, the code of the domain classes is generated following a concrete strategy. It is important to remark that this strategy to generate code is not attached to any particular programming language. This is possible because the final implementation is obtained in an automated way by programmingthe corresponding mappings between conceptual model constructs and their representation in a particular software development environment (Java in this paper).

Having obtained the full object-oriented implementation, the next step is how to obtain a component specification of the system.

4.4 From problem domain classes to business components.

Starting from the code generated for the domain classes, an interface specification of the system can be obtained in an automated way. This interface specification has two benefits:

- the application generated can be execute in a distributed environment.
- the components obtained can be reusable for other applications.

An interface definition language (IDL), is used to define the interface of the domain classes in a programming language-neutral form. We are going to illustrate this process for the example of the class reader written in Java. In the context of Java environment, Java RMI can be used. One of the main benefits in the use of RMI to define the interface is that it uses the actual Java interface definition. In consecuence, no external languages are needed to write the interface.

Figure 12 shows the interface definition for the class reader. The import statement imports the Java RMI package. The interface I_reader is a normal Java interface with two interesting characteristics:

- It extends the RMI interface named Remote, which marks the interface as one available for remote invocation.
- All its methods throw RemoteException, which is used to signal network and messaging failures.

The interface itself supports one method: activateService which returns a boolean value that indicates is a particular service has could been executed.

```
// I_reader.java - Definition of the interface
import java.rmi.*;
public interface I_reader extends Remote {
    boolean activateService(String service,
    String objectIdentifier, Hastable parameters)
    throws ActivateServiceException;
}

// reader.java  - Implementation of the interface
public class reader extends UnicastRemoteObject
implements I_reader {...}
```

Figure 12: interface definition for the reader class

The Java class reader[3], implements the method of the remote interface Ireader. In this way, clients on remote hosts can use RMI to send messages to reader objects.

Following this strategy, a client program, uses a lookup operation consist of an URL string indicating the name under a RMI object is bound in the registry. It results in the stub of the bound RMI object to be shipped to the client. From this moment, the client can make remote calls to the server.

In this way, an interface definition serves as a contract between the client of the interface and the provider of an implementation of the interface. Having obtained the interface definition for each domain class, they are grouped in a package to build the component specification of the application logic.

If we are not dealing with Java. An external IDL must be used following the same strategy. Finally, this component specification, once compilated can be deployed in the proposed architecture.

5 Conclusions

Over the past few years, constructing applications by assembling reusable software components has emerged as highly productive and widely accepted way to develop custom applications. In this paper we have presented a model and an component-based architecture for this model that allow developers to generate software components starting from the conceptual modeling step. These components can be combined to build a software prototype that is functionally equivalent to the specification collected. To achieve this goal, we use a well-defined OO-Method methodological framework, which properly connect OO conceptual modeling and OO software development environments using a component-based model. The most relevant contributions of this paper are the following:

[3]The type UnicastRemoteObject defines the kind of remote object the server will be, in this case a single server as opposed to a replicated service.

- A process to support Component-based development in the OO-Method methodological approach.
- A concrete strategy to generate code based on a clearly separation of component specification from component implementation to enable technology-independent application design.
- A well-defined component-based framework to execute the specification in the solution space.

These ideas are being applied in the context of a CBD tool that has been called JEM. JEM is a Java-based implementation for the OO-Method Execution Model. The basic purpose of JEM is to animate conceptual models captured with the OO-Method CASE Tool, over ditributed internet/intranet environments. JEM fulfill the requirements respect to the model and the architecture proposed in this paper for the execution model. A beta version of JEM can be reached at: http://oomethod.dlsi.ua.es:8080/jem.

References

[1] G. Booch. *Object Oriented Analysis and Design with Applications. Second Edition.* Addison-Wesley, 1994.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. UML v1. Technical report, Rational Software Corporation, 1997.

[3] D. Box. *Creating Components with DCOM and C++.* Addison-Wesley, 1997.

[4] AW. Brown. From Component-based infrastructure to Component-based Development. In AW. Brown and K. Wallnau, editors, *2nd. International Workshop on Component-based Software Engineering*, pages 87–93, 1998.

[5] K. Brown and B. Whitenack. *Crossing Chasms. Pattern Languages of Program Design vol. 2.* Addison-Wesley, 1996.

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

[7] TB. Downing. *Java RMI: Remote Method Invocation.* IDG Books, 1998.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Desing Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[9] Object Management Group. The Common Object Request Broker: Architecture and Specification. Technical report, Object Management Group, 1997.

[10] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *OO Software Engineering, a Use Case Driven Approach.* Addison-Wesley, 1992.

[11] O. Pastor, F. Hayes, and S. Bear. OASIS: An Object-Oriented Specification Language. In P. Loucopoulos, editor, *Advanced Information Systems Engineering*, volume 593 of *Lecture Notes in Computer Science*, pages 348–363. Springer-Verlag, 1992.

[12] O. Pastor, E. Insfrán, J. Merseguer, J. Romero, and V. Pelechano. OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods. In A. Olivé and JA. Pastor, editors, *Advanced Information Systems Engineering*, volume 1250 of *Lecture Notes in Computer Science*, pages 145–158. Springer-Verlag, June 1997.

[13] O. Pastor, E. Insfrán, V. Pelechano, and J. Gómez. From Object Oriented Conceptual Modeling to Automated Programming in Java. In Tok Wang Ling, Sudha Ram, and Mong Li Lee, editors, *17th International Conference on Conceptual Modeling*, volume 1507 of *Lecture Notes in Computer Science*, pages 183–196. Springer-Verlag, November 1998.

[14] O. Pastor and I. Ramos. *OASIS 2.1.1: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*. Servicio de Publicaciones. Universidad Politécnica de Valencia, tercera edición edition, 1995.

[15] NW. Paton and O. Díaz. Active Database Systems. *ACM Computing Surveys*, 1998.

[16] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice-Hall, 1991.

[17] R. Schulte. Three-tier computing architectures and beyond. Technical report, Gartner Group, 1995.

[18] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, March 1992.

[19] RJ. Wieringa. *Algebraic Foundations for Dynamic Conceptual Models*. PhD thesis, Vrije Universiteit, Amsterdam, 1990.

[20] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object Oriented Software*. Prentice-Hall, 1990.