From Object Oriented Conceptual Modeling to Automated Programming in Java

Oscar Pastor, Vicente Pelechano, Emilio Insfrán, Jaime Gómez

Department of Information Systems and Computation

Valencia University of Technology

Camí de Vera s/n

46071 Valencia (Spain)

{ opastor | pele | einsfran | jgomez }@dsic.upv.es

Abstract

The development of Internet commercial applications and corporate Intranets around the world, which uses Java as the *de facto* standard language, is a significant topic in modern Software Engineering. In this context, more than ever, well-defined methodologies and high-level tools are essential for developing quality software in a way that should be as independent as possible of the changes in technology.

In this article, we present an OO methodology based on a formal object-oriented model. The main feature of this method is that developers' efforts are focused on the conceptual modeling step, where analysts capture system requirements, and the full implementation can automatically be obtained following an execution model (including structure and behaviour). The final result is a web application with a three-tiered architecture, which is implemented in Java with a relational DBMS as object repository.

Keywords: Conceptual modeling for information systems. Rapid prototyping. CASE tools. Integrating data and behaviour modeling, Internet development

From Object Oriented Conceptual Modeling to Automated Programming in Java

1. Introduction

The boom of environments commonly known as “web computing” environments, has originated the development of Internet commercial applications and the creation of corporate Intranets around the world. The use of the Java language [1] in these environments has opened a significant research work related with how to implement properly a final correct software product. In this context, where new technologies are continuously emerging, the software development companies must make suitable methods, languages, techniques and tools for dealing with the new market requirements. More than ever, well-defined methodologies and the high-level tools which support them are essential for developing quality software in a way that should be as independent as possible of the changes in technology. The idea of clearly separating the conceptual model level, centered in *what* the system is, and the execution model, intended to give an implementation in terms of *how* the system is to be implemented, provides a solid basis for operational solutions to the problem. If we have rich-enough conceptual modeling environments to capture the relevant system properties in the problem space, a right software representation at the solution space can be easier generated.

Nowadays OO methodologies like OMT [2], OOSE [3] or Booch [4] are widely used in industrial software production environments. Industry trends attempt to provide unified notations such as the UML proposal ([5]) which was developed to standardize the set of notations used by the most well-known existing methods. Even if the attempt is commendable, this approach has the implicit danger of providing users with an excessive set of models that have overlapping semantics without a methodological approach. Following this approach we have CASE tools such as Rational ROSE/Java [6] or Paradigm Plus [7] which include Java code generation from the analysis models. However if we go into depth with this proposed code generation feature, we find that it is not at all clear how to produce a final software product in Java which is functionally equivalent to the system description collected in the conceptual model. This is a common weak point of these approaches. Far from what is required, what we have after completing the conceptual model is nothing more than a template for the declaration of classes where no method is implemented and where no related architectural issues are taken into account.

In order to provide an operational solution to the above problem, in this paper we present a method, which is based on a formal object-oriented model. The main feature of this method is that developers' efforts are focused on the conceptual modeling step, where analysts capture system requirements, and the full implementation can automatically be obtained following an execution model (including structure and behaviour). The final result is a web application with a three-tiered architecture, which is implemented in Java with a relational DBMS as object repository.

A CASE Tool gives support to this method[1]. It constitutes an operational approach to the ideas of the *automated programming paradigm* [8]: a collection of system information properties in a graphical environment (*conceptual modeling step*), automated generation of a formal OO system specification (used as system repository) and of a complete software prototype (including statics and dynamics) obtained from the conceptual model and which is functionally equivalent to the quoted system specification (*execution model step*).

2. The OO-Method: An Object-Oriented Methodology

Following the OO-Method strategy, the software production process starts with the conceptual modeling step where we have to collect the relevant system properties. Once we have an appropriate system description, a formal OO specification is automatically obtained. This specification is the source of a well-defined execution model which determines all the implementation-dependent features in terms of user interface, access control, service activation, etc. This execution model provides a well-structured framework that enables the building of an automatic code generation tool. It is important to note that the formal specification is hidden to the OO-Method user: the relevant system information is introduced in a graphical way, which is syntactically compliant with the conventional OO models, but which is semantically designed to fill the class definition templates according to the formal OO basis.

2.1 OASIS: an Object-Oriented Formal Model

The OO-Method was created on the formal basis of OASIS, an OO formal specification language for Information Systems [9]. In fact, we can see OO-Method as a graphical OASIS editor, built using the conventional approach of using an object, dynamic and functional model to make designers think that they are using a conventional OO method. The formalism is in this way hidden to them. Below, we give a quick overview of the characteristics of OASIS.

From an intuitive point of view, an object can be viewed as a cell or capsule with a state and a set of services. The state is hidden to other objects and can be handled only by means of services. The set of services is the object's interface, which allows other objects to access the state. Object evolution is characterized in terms of changes of states. Events represent atomic changes of state and can be grouped into *transactions*[2].

When we build a system specification, we specify classes. Classes represent a collection of objects sharing the same *template*. The template must allow for the declaration of an identification mechanism, the signature of the class including attributes and methods, and finally a set of formulae of different kinds to cover the rest of the class properties:

---

[1] The OO-Method CASE tool development has been partially supported by a grant from the IMPIVA as a project of the Valencia Plan of Science and Technology (Spain).

[2] Molecular units of processing composed of object services that have the properties of non-observability of intermediate states and the all-or-nothing policy during execution.

- integrity constraints (static and dynamic) which state conditions that must be satisfied.
- valuations which state how attributes are changed by event occurrences.
- derivations which relate some attribute values to others.
- preconditions which determine when an event can be activated.
- triggers which introduce internal system activity.

Finally, as an object can be defined as an observable process, a class definition should be enriched with the specification of the process attached to the class. This process will allow us to declare possible object lives as *terms* whose elements are events and transactions. OASIS deals with complexity by introducing aggregation and inheritance operators. A complete description of the OASIS language can be found in [10].

2.2 Conceptual Modeling

Conceptual modeling in OO-Method collects the Information System relevant properties using three complementary models:

- **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).

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

  - *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.

  - *Object Interaction Diagram:* represents interobjectual 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.

- **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 that allows us to generate a complete OASIS specification in an automated way. More detailed information can be found in [11].

From these three models, a corresponding formal and OO OASIS specification is obtained using a well-defined translation strategy. The resultant OASIS specification acts as a complete high-level system repository.

2.3 Execution Model

Once all the relevant system information has been specified, we use an execution model 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 all the system properties in a logical three-tiered architecture for any target software development environment:

- **interface tier**: classes that implement the interaction with end users presenting a visual representation of the application and giving users a way to access and control the object's data and services.

- **application tier**: classes that fully implement the behaviour of the business classes specified in the conceptual modeling step enforcing the semantics of our underlying object model

- **persistence tier**: classes that provide services allowing the business objects to interact with their quoted permanent object repository.

In order to easily implement and animate the specified system, we predefine a way in which users interact with system objects. We introduce a new way of interaction, close to what we could label as an OO virtual reality, in the sense that an active object immerses in the object society as a member and interacts with the other society objects. To achieve this behaviour the system has to:

1. **identify the user** (an access control): logging the user into the system and providing an **object system view** determining the set of object attributes and services that it can see or activate.

2. **allow service activation**: finally, after the user is connected and has a clear object system view, the users can activate any available service in their worldview. Among these services, we will have system observations (object queries) or events or transactions served by other objects.

The process of access control and the building of the system view (visible classes, services and attributes to the user) are implemented in the interface tier. The information to properly configure the system view is included in the system specification obtained in the conceptual modeling step.

Any service activation has two steps: build the message and execute it (if possible).

In order to build the message the user has to provide information to:

1. **identify the object server**: The server object existence is an implicit condition for executing any service, unless we are dealing with a new event[3]. At this point, the persistence tier retrieves the object server from the database.

2. **introduce event arguments**: The interface tier asks for the arguments of the event being activated (if necessary).

Once the message is sent, the service execution is characterized by the occurrence of the following sequence of actions in the server object at the application tier:

1. **check state transition**: verification in the object State Transition Diagram (STD) that a valid *transition* exists for the selected service in the current object state.

2. **precondition satisfaction**: the precondition associated to the service must hold

   If 1 and 2 don't hold, an exception will arise and the message is ignored.

3. **valuation fulfillment**: the induced event modifications (specified in the Functional Model) take place in the involved object state.

4. **integrity constraint checking in the new state**: to assure that the service execution leads the object to a valid state, the integrity constraints (static and dynamic) are verified in the final state. If the constraint does not hold, an exception will arise and the previous change of state is ignored.

5. **trigger relationships test**: 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.

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 a programming environment.

3. An Architecture for Implementing the Execution Model in Java

The abstract execution model shown above is based on a generic three-tiered architecture. Below, we will introduce a concrete implementation using web technology and Java as the programming language. This will provide a methodological framework to deal with Java implementations, which are functionally equivalent to the source conceptual model.

[3] Formally, a new event is a service of a metaobject that represents the class. The metaobject acts as object factory for creating individual class instances. This metaobject (one for each class) has the class population attribute as a main property, the next *oid* and the quoted new event.

3.1 Translating a Conceptual Model into Java Classes using the Execution Model

Starting from the proposed Execution Model, we want to design the architecture of classes needed to implement the three logic levels: interface, application and persistence. Following, we specify the most relevant features of the Java classes needed to support the intended architecture:

At the **interface level** we have complementary classes, which are not explicitly used in the conceptual model but which help to implement the interaction between the user and the application, following the underlying object model semantics.

- *Access_control class*. This class extends a *panel* with the typical widgets to allow users to be identified as a member of the object society (by providing the object identifier, password and the class to which the user belongs to). One *access control object* is created every time that a user wants to be connected to the system as an *active* object sending and receiving messages. This class implements the first step of our execution model.

```
import java.awt.*;
import java.lang.*;
import excepciones.*;
public class Access_control extends Panel { ... }
```

- *System_view class*: Once an active object (user) is connected to the system, a *system_view object* (instance of the system_view class) will show a page with the same number of items as classes the user is allowed to interact with. These items are clickable regions that the user can activate in order to see the services (also displayed as clickable items) that he/she can use. The declaration of the class is the following:

```
import java.applet.*;
import java.awt.*;
import excepciones.*;
public class System_view extends Applet implements Runnable { ... }
```

- *Service_activation class*: This class will define a typical web interface for data entry, where the relevant service arguments are requested. The service_activation object will be a generic one for all the services of all the classes. Depending on the service activated, it will show the corresponding edit boxes for the identification of the object and the parameters of the service. Once they are fulfilled, the user can send the message to the destination (submit) or ignore the data request action (cancel).

```
import java.awt.*;
public class Service_activation extends Panel { ... }
```

At the **application level** we have the classes that implement the behaviour of the business classes specified in the conceptual model.

In order to ensure that the implementation of the application classes will follow our underlying object model semantic and have persistence facilities, we will define our business classes as an implementation of an OASIS interface and an extension of an Object_mediator class [12].

The OASIS interface specifies the necessary services to support the execution model structure, as shown in the following paragraph and the Object_mediator class that is explained at the next level:

```
import java.awt.*;
interface Oasis {
       void check_precondition (string event)
       void check_state_transition (string event)
       void check_integrity_constraint ()
       void check_trigger ()
       ...
}
```

At the ***persistence level*** a Java class called *object_mediator* must be created. It implements the methods for saving, deleting and retrieving system domain objects that are stored in a persistent secondary memory (object repository). The *object_mediator* class has the following general structure:

```
class Object_mediator {
       void delete ();
       void save ();
       void retrieve ();
}
```

JDBC classes will be used for proper interaction with the involved RDBMS servers in the implementation of these methods.

Even if we focus on Java in this paper, this design could be translated to any other OO programming language by properly distributing the application components depending on the target environment characteristics and necessities.

3.2 Distributing Java classes in a Web Architecture

Once the previous components have been created, they must be properly distributed in a web architecture. Many proposals to distribute Inter/Intranet application components exist. We are going to present a three-tiered architecture (see Figure 1) which fits very well with the OO-Method Execution Model features presented above.

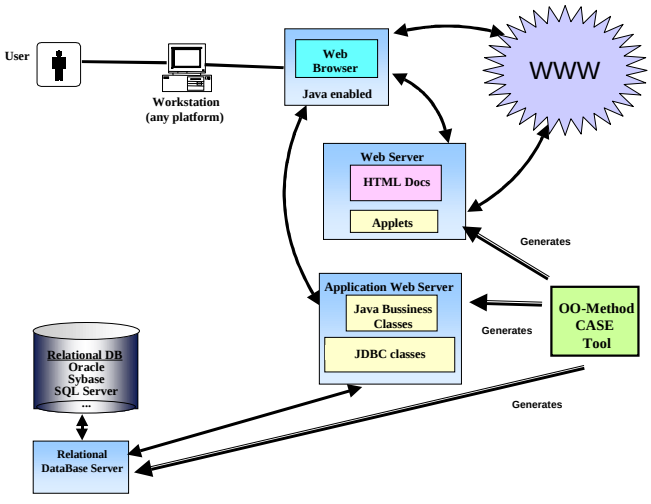

Figure 1. Three-tiered web architecture

A client (a web browser) unloads the relevant HTML pages from a *web server*, together with the applets that conform the application interface. The user will interact with the Java system objects through the web client. These Java system objects will be stored in a *web application server* that will query or update the object state stored in a *Data Server*, through the services provided by the JDBC objects.

It is important to remark again that the components distributed in this architecture are generated in an automated way using the OO-Method CASE tool. This is done by defining a precise mapping between the finite set of *behavioural patterns* that constitutes the Conceptual Model in OO-Method and their corresponding software representation in the target software development environment, Java in this work. Preconditions, state transitions, integrity constraints and triggers are declared in the graphical OO-Method models, and implemented in each one of the *Java Business Classes* as was presented before. As they finally are well formed formulas in OASIS, these formulas are translated into the Java syntax. This is how the problem space concepts are converted into their corresponding software representation. Let's show clearer all these concepts through an example.

4. Automated Programming in Java. A Case Study

A CASE Tool supporting the OO-Method allows us to model and automatically generate fully functional prototypes in Java. In order to better understand the architecture and behaviour of the previous component classes generated using the Java Execution Model, we introduce a Rent-a-Car case study as a brief example:

> "A company rents vehicles without drivers. These vehicles are bought at the beginning of the season and usually sold when the season is over. When a customer rents a vehicle, a contract is generated and it remains open until the customer returns the vehicle. At that time, the total amount to be paid is calculated. After this step, the Vehicle is ready to be rented again".

First, we construct the Conceptual Model (object, dynamic and functional models) by identifying the classes and specifying their relationships and all the static and dynamic properties. We cannot present due to space limitations the three OO-Method models (object, dynamic and functional) of this simple example, but let's assume that the classes identified in this problem domain are the following: *Contract*, *Customer*, *Vehicle* and *Company*. Every class will basically have its corresponding set of attributes and services, and the declaration of preconditions, integrity constraints, valid transitions, evaluations and triggers.

Based on the Execution Model proposed above, we obtain the architecture of the web application in an automated way. This architecture includes Java classes and relational tables attached to the conceptual model in Figure 2.

Persistence Tier	Application Tier	Interface Tier
Customer table	Customer class	Access_control class
Vehicle table	**Vehicle class**	System_view class
Contract table	Contract class	Service_Activation class
...	...	...

Figure 2. Rent-a-Car system implementation architecture

The Java code that implements a *business class* in the application tier (Vehicle class) following the execution model strategy can be seen in Figure 3, where comments have been introduced with the aim of making it self-explanatory.

Following, we are going to describe an illustrative scenario for the Rent-a-Car prototype generated. This scenario will show the interaction between Java objects and their behaviour when a user enters into the object system.

When a client loads the main HTML page that calls the Java applet, an instance of the ***Access Control*** class is created and *User identification* is required (see Figure 4).

```
package application_tier;

import exceptions.*;
import object_mediator.*;
import oasis.*;

public class Vehicle extends Object_mediator
       implements Oasis {
       // Attributes specified in the conceptual model class Vehicle
       private string state;
       ...
       // Events specified in the conceptual model
       // The following method implements the change of object's state
       protected boolean rent() throws EX_Check_Error
       {   state="rented";
       ...
       }
       ...
       // The following method implements the precondition checking
       protected void check_precondition(string event) throws EX_Check_Error
       {...}

       // The following method implements the execution of the rent event
       public void eval_rent()
       {
         try   {
               retrieve( );       // retrieves the object from the Database
               check_precondition('rent');
               check_state_transition('rent');
               rent();
               check_integrity_constraint();
               check_triggers();
               save( );           // saves the object in the Database
               }
               catch(EX_Check_Error e) {...}
       }
       ...

       }
```

Figure 3. Automatically generated Java code for the vehicle class

After a *User* connects to the *Rent-a-Car* system, a *menu page* with an option for every class will appear (see Figure 5). If the user clicks in a class option, a new *menu page* associated with the selected class will be generated (including one option for every class event or transaction).

Figure 4. A User Access Control Page

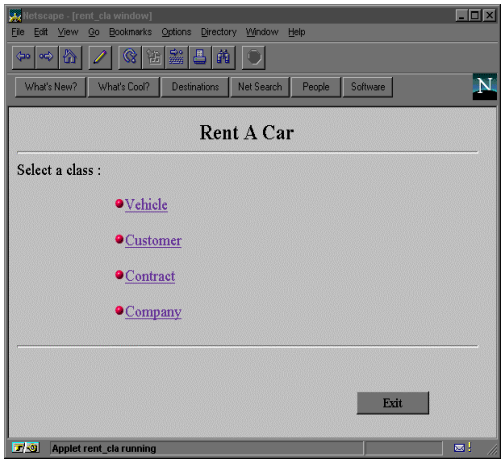

Figure 5. Rent-a-Car System View page.

In our example, when the user selects the *Vehicle* class, the service list offered by this class is shown, as can be seen in Figure 6.

Every service option activation will generate a new *request parameter page*, as can be seen Figure 7. This page will ask the user for the arguments needed to execute the service. The *Ok control button* has a code associated to it that will call to a class method that implements the effect of the service on the object state. This method will check the state transition correctness and method preconditions. If this checking process succeeds, the object change of state is carried out according to the *functional model* specification.

We finish the method execution by verifying the integrity constraints and the trigger condition satisfaction in the new state. Object state updates in the selected persistent object system become valid through the services inherited from the *object_mediator* class.

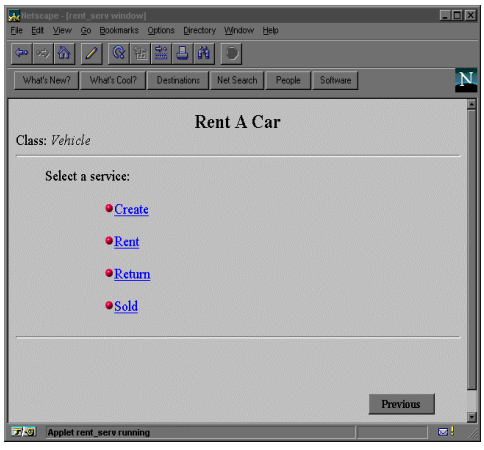

Figure 6. Vehicle Class Services Menu Page.

Figure 7. A Parameter Request Page.

5. Conclusions and further work

Advanced web applications will depend on object technology to make them feasible, reliable and secure. To achieve this goal, well-defined methodological frameworks, which

properly connect OO conceptual modeling and OO software development environments, must be introduced. The OO-Method provides such an environment. The most relevant features are the following:

- an operational implementation of an automated programming paradigm where a concrete execution model is obtained from a process of *conceptual model translation*

- a precise object-oriented model, where the use of a formal specification language as a high-level data dictionary is a basic characteristic

All of this is done within the next-generation web development environments, making use of the Inter/Intranet architectures and using Java as a software development language.

Research work is still being undertaken to improve the quality of the final software product that is generated including advanced features such as user-defined interface, schema evolution or optimized database access mechanisms.

References

[1] Arnold K., Gosling J. *The Java Programming Language*. Sun MicroSystems. Addison-Wesley, 1996.

[2] Rumbaugh J. et al. W. *Object Oriented Modeling and Design*. Englewood Cliffs, Nj. Prentice-Hall. 1991.

[3] Jacobson I. et al. G. *OO Software Engineering , a Use Case Driven Approach*. Reading, Massachusetts. Addison -Wesley.

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

[5] Booch G., Rumbaugh J., Jacobson I. *UML. v1*. Rational Software Co., 1997.

[6] Rational Software Corporation. Rational Rose User's Manual, 1995.

[7] Platinum Technology, Inc., Paradigm Plus: Round-Trip Engineering for JAVA, White Paper from Platinum Web Site: http://www.platinum.com/. 1997.

[8] Balzer R. et al. *Software Technology in the 1990s: Using a New Paradigm*. IEEE Computer, Nov. 1983.

[9] Pastor O.,Hayes F. and Bear S. OASIS: An object-oriented specification language. In P. Loucopoulos, editor, Proceedings of the CAiSE'92 conference, pp. 348-363, Berlin, Springer, LNCS 593 (1992).

[10] 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).

[11] Pastor O. et al. *OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods*. In Antoni Olivé and Joan Antoni Pastor editors, Proceedings of CAiSE97 conference, pp. 145-158 ,Berlin, Springer-Verlag, LNCS 1250. June 1997.

[12] Argawal S., Jensen R., and Keller A. M. Architecting object applications for high performance with relational databases. In OOPSLA Workshop on Object Database Behaviour, Benchmarks, and Performance, Austin, 1995.