A Framework to Translate UML Class Generalization into Java Code*

Pedro S nchez[1], Patricio Letelier[2], Juan A. Pastor[1], and Juan A. Ortega[3]

[1]Universidad Polit cnica de Cartagena, Spain
{pedro.sanchez,juanangel.pastor}@upct.es
[2]Universidad Polit cnica de Valencia. Spain
letelier@dsic.upv.es
[3]Universidad de Sevilla. Spain
ortega@lsi.us.es

Abstract. The concept of generalization used during analysis when building a class diagram has a close relationship with the notion of inheritance included in object-oriented programming languages. However, from the point of view of programming, inheritance is a useful mechanism but not especially conceived to implement the generalization specified in analysis. Thus, generalization should be treated suitably in order to obtain sounded design and code from analysis specifications. In addition, it is known that it does not exist concensus about the interpretation and use of inheritance and each programming language provides its particular vision. Hence, when moving from analysis to design and/or implementation (and normally without using a formal approach) the generalization relationships are prone to misinterpretation. OASIS is a formal approach to specify object-oriented conceptual models. In OASIS generalization is included as a language construct that allows specifying generalization patterns with precise semantic and sintaxis. Although OASIS is a textual formal language, the main aspects of one OASIS specification can be mapped and represented using the UML notation, in particular generalization relationships among classes. In this paper we present OASIS generalization patterns and we show how they can be implemented in Java. We also propose other ways to carry out this implementation.

1 Introduction

Generalization is a specification mechanism that allows introducing taxonomic information in the model of a system. Generalization improves the reusability and extensibility of specifications. Class generalization establishes an ordering between classes, *parent classes* and *child classes*. Child classes inherit structure and behavior

* This work has been supported by CICYT (Project DOLMEN-SIGLO) TIC2000-1673-C06-01.

Z. Bellahs ne, D. Patel, and C. Rolland (Eds.): OOIS 2002, LNCS 2425, pp. 173-185, 2002.

from parent classes. In UML[1] [1], as in other previous modeling notations, class generalization[2] has been roughly defined, mainly because there is no wide consensus about its interpretation and usage [2]. The inheritance mechanism behind the generalization relationship has two points of view: it is a modelling tool and it is a code reuse tool, these visions are usually at odds [3]. Due to the lack of precise semantics for generalization at the conceptual level, usually when building analysis models[3] it is implicitly assumed the interpretation of the target object-oriented language which normally it is not the most suitable perspective. In these circumstances the analyst can specify generalization relationships at implementation level which is obviously a contradiction respect to the level of abstraction of analysis. Usually, the solution is to leave generalization relationships loosely defined, which has the risk of possible misinterpretation in design and implementation. In addition, when using semi-formal methods and notations the transition from models to its sounded implementations is even more difficult.

As far as generalization is concerned, UML establishes that the more specific element is fully consistent with the more general element (it has all its properties, members, and relationships) and may contain additional information . Wegner proposed four levels of compatibility between the child class and the parent class, in a descending order of compatibility they are [4]: behavior compatibility, signature compatibility (the same signature for the same operations), name compatibility (the same names for the same operations) and cancelation (some parent class operations are not available in the child class). At the implementation stage, inheritance between classes (the implementation of generalization) means adding, redefining or canceling the inherited features and associations in the child class. This wide definition has different interpretations in distinct programming languages. Thus, in general, it depends on the programmer to maintain the correspondences between the implementation of the child class and its corresponding parent class.

OASIS (Open and Active Specification of Information Systems) [5] is a formal approach to conceptual modeling following the object-oriented paradigm. Generalization in OASIS is included as a language constructor which allows specifying the different aspects characterizing a generalization relationship between classes. Thus several common patterns of generalization can be directly represented with OASIS. Although OASIS is a textual language, most parts of a specification can be represented using UML, in particular generalization relationships. The aim of this paper is to show how to use the OASIS generalization as the intended semantic for class generalization in UML models. Thus analysis specifications with generalization

[1] In this work we have used the UML specification version 1.4.

[2] Previously to UML this term was commonly referred as class specialization. Both generalization and specialization are abstraction mechanism, used to establish inheritance relationships between classes. The nuance between generalization and specialization is whether, taking the parent classes we establish the child classes (specialization), or taking the child classes we establish the parent class (generalization). But in the end, the result is the same: there will exist inheritance from the parent class to the child class.

[3] We have preferred using analysis to conceptual modeling (and analysis model to conceptual model) to follow the UML terminology.

can have a seamless and sounded translation to design and implementation. To illustrate the implementation aspects we will use Java.

The organization of this article is as follows. After this introduction section, we briefly introduce the OASIS formal framework for generalization. Next, we give a description of the equivalences between the UML generalization and the Java programming language. Then, we comment some related works and finally, we present the conclusions and future work.

2 Generalization in OASIS

Through generalization a new class specification can be partially defined establishing a relationship with others already defined classes. The new class can summarize or extend the previous classes. The generalization incorporated in OASIS is based on the work by Wieringa et Al. [6]. In a generalization relationship one class play the role of *parent class* and the others are *child classes*. A child class inherits the features defined in the parent class. A child class can be at the same time parent class, and in this form, a hierarchy of classes is created (it is a directed acyclic graph). The features that are common to all child classes are specified in the parent class, and the specific child class features are defined in the corresponding child classes. In OASIS we distinguish three orthogonal kinds of generalizations: *static classification*, *dynamic classification* and *role group*.

Each of these kinds of generalization with its associated characteristics represents a different conceptual modeling pattern of generalization, offered directly as OASIS constructs. With this expressiveness the modeling task can focus on specifying the problem rather than on its solution in a specific programming language. At the same time these patterns of generalization impose a disciplined usage of inheritance [7], constraining its utilization only to specify relevant aspects at the level of analysis.

Although different aspects of OASIS are normally presented using several convenient and more expressive formalisms, Dynamic Logic constitutes the basic and uniform formal support. In this article we present the generalization in OASIS using set theory and process algebra. However, we have established the corresponding mappings to Dynamic Logic [8].

In this paper we will focus only on static and dynamic classifications due to the fact that these are the only kinds of generalization defined in UML. Role groups are a more specific kind of generalization useful when a inheritance relationship is necessary but the object in the child class is a role of the object in the parent class (the player), they are thus different objects. Though not so directly, role groups can be modeled using association relationships.

2.1 Classification Hierarchies

A classification hierarchy establishes an inheritance relationship among one parent class and child classes whose populations are subsets of the parent class instances. For each class C, two aspects can be distinguish:

- The class **intention**, int(C), is the set formed by all the class features and associations. It represents the class type.
- Given any instant t, the class **extension** (population), $ext_t(C)$, is the set of instances of the class in that instant.

Let C_1 and C_2 be two classes, if $ext_t(C_1) \subseteq ext_t(C_2)\ \forall t$, then C_1 is child class of a classification of C_2. When this occurs, a inverse inclusion relationship between the intensions of C_1 and C_2, that is, $int(C_2) \subseteq int(C_1)$.

A classification hierarchy divides[4] the population of the parent class in **disjoint** subsets. That is, let $C_1, \ldots, C_n$ be child classes of the parent class C_0. Then

$$ext_t(C_0) = \cup\ ext_t(C_i)\ .$$
$$ext_t(C_i) \cap ext_t(C_j) = \varnothing,\ i \neq j \neq 0\ . \tag{1}$$

Demanding that each classification hierarchy be complete and disjoint solves several ambiguities and contradictions. Let us consider the example where the class `student` is child class of `person`. A person object can be or not a student object in one instant of its existence. The operation `become_student` should occur in the life of a `person` object and not in the life of a `student` object[5], but class `student` inherits this operation, thus this is contradictory. This problem is solved by having complete classifications, for example in this case specifying the child class `not_student` and instead of putting the operation `become_student` in the class `person`, put it in the class `not_student`.

In a classification hierarchy each object is an instance of the parent class and at the same time of one (and only one) of the child classes, that is, it is the same object (the same Oid). This dictates the following semantic:

- There must exist behavior compatibility between the child class and the parent class, that is, the Substitution Principle [9] must be accomplished. Thus, every object instance of the child class could be used as an object instance in the context of the parent class.
- When the object in the parent class is destroyed, this also means its destruction in the child class, and vice versa too.

2.2 Static Classification

In a static classification the object instances of the child classes are associated to them during their whole lives. That is, suppose t_1 and t_2 any two instants, C_i and C_j $(i \neq j)$ child classes of a static classification. Then:

$$ext_{t1}(C_i) \cap ext_{t2}(C_j) = \varnothing\ . \tag{2}$$

Fig. 1 shows two static classifications of the class vehicle. The discriminators by fuel and by purpose allow us to distinguish two different hierarchies with the same parent class.

---

[4] Although this does not mean that in UML we are obliged to specify all the child classes. The constraint *incomplete* can be attached to the classification hierarchy indicating that there is an implicit child class others .

[5] Except if we want to specify an OASIS role group hierarchy which does not exist in UML.

2.3 Dynamic Classification

In a dynamic classification hierarchy the instances of one child class can migrate to another child class in the hierarchy. Thus the intersection presented in formula 2 can be different from the empty set. There are at least two ways of specifying the migratory process: on the one hand based on the occurrence of certain actions, on the second hand based on the state of the object (the values of its attributes).

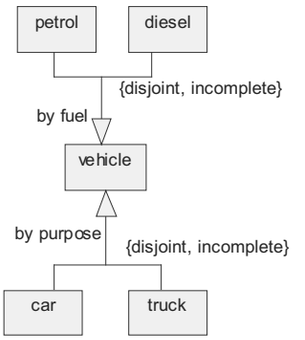

Fig. 1. Two static classifications of `vehicle`

An important difference between static classification and dynamic classification is that in the later the extension of one of its child classes can change without changing the extension of the parent class. It is not allowed specifying a static classification by taking as a parent class a class that is child class in a dynamic classification. This constraint eliminates unnecessary complexity in the model, without decreasing the expressiveness of the language.

Dynamic Classification based on event occurrence. In this case the migratory process is defined by means of a process specification[6]. The operations involved in the process specification belong to the child class where the migration step takes place. The agent constants (or states in a state diagram) are the names of the child classes. By default, the starting state receive the name of the parent class and the new event establishes the first transition to the suitable child class. A dynamic classification of `car` determined according the occurrence of events `new_car`, `be_repaired` and `break_down` is shown in Fig. 2.

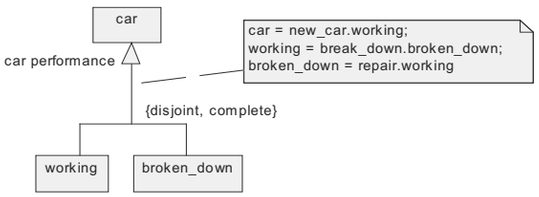

Fig. 2. A dynamic classification of `car` based on action occurrence

[6] In OASIS we use a simple process algebra. It would be a state diagram in UML.

The creation of one instance of `car` class implies that its life starts belonging to the child class `working`. While it belongs to the class `working` it can be affected by events of the parent class `car` or child class `working`, for example, the event `break_down`. If this event occurs, then the instance migrates from the child class `working` to the child class `broken_down`.

Dynamic Classification based on the state of the object. In this case the migratory process is determined by the state of the object. Each time that the object reaches a new state (a new set of values for its attributes) this can involve its migration from one child class to another in the dynamic classification. A dynamic classification of class `account` based on the values of the attribute `balance` is shown in Fig. 3. In this example, the initial child class for a new `account` object will be established according to the initial value of the attribute `balance`.

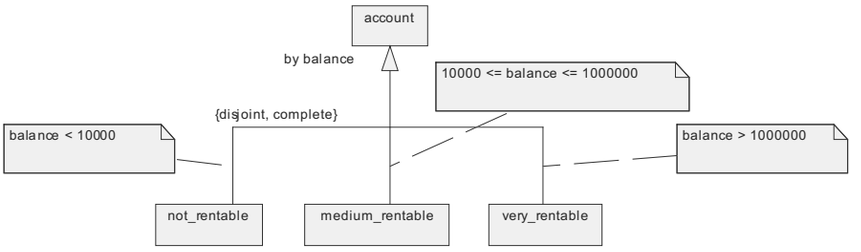

Fig. 3. A dynamic classification of `account` based on the state of the object

2.4 Species and Multiple Classification

A species is basically a class whose type is obtained combining the types of the child classes in the lower levels of a hierarchy of classes, taking each child class from a different classification. Each species involves the notion of multiple inheritance of the features and associations belonging to each child class selected. Species cannot be child class of any parent class what is normally allowed in UML and programming languages. Thus, this concept of species involves a disciplined use of multiple classification which has methodological advantages due to the fact that the model is clearer and prevents mistakes [6]. In example, the class (species) `truck*diesel` can have as an emergent property the attribute representing the date of the latest revision. Thus we specify this class in the same way as a non-species class, defining the emergent attribute.

An object instance of the species `broken_down*car*petrol` is an instance of each of the selected child classes. Thus, multiple classification is the situation when a child class inherits features and associations from more than one parent class. In the OASIS context a species is a child class participating in multiple classification. The emergent features or associations that a species can have are specified in a usual class specification. If there are not emergent features or associations it is not necessary to explicitly specify the species class.

3 Java Implementation

Regarding the implementation in Java language, the class generalization framework of OASIS demands four features that are not offered directly or imposed in Jav

1. There must be supported dynamic classification.
2. It might be more than one static or dynamic classifications (distinct classification hierarchies) with the same parent class.
3. Every classification must have at least two child classes.
4. Method implementation in child classes must accomplish behavior compatibility with the method implementation in the parent class.

Instead of giving a general implementation pattern we have developed implementation solutions for representative situations and it should be easy to extrapolate to other more specific cases. We will call *simple* the situation in which for the same parent class exists only one classification (static or dynamic), otherwise we will call it *complex* (where multiple inheritance by means of *species*).

3.1 Static Classification (Simple)

Let us consider the next dynamic classification where class `C` is the parent class[7]:

```
C1, C2 static specialization of C;
C11, C12 static specialization of C1;
```

In static classifications objects are created at the lowest class in the hierarchy (leaf classes). In the example these classes are `C11`, `C12` and `C2`. The Java implementation would be:

```
public abstract class C {…}
public abstract class C1 extends C {…}
public class C2 extends C {…}
public class C11 extends C1 {…}
public class C12 extends C2 {…}
```

The classes C and C1 are labeled abstract. Thus, it is not possible to create objects directly on them. The following object creations are allowed:

```
C11 objC11 = new C11(…);
C12 objC12 = new C12(…);
C2 objC2 = new C2(…);
```

3.2 Dynamic Classification (Simple)

Regarding dynamic classification, Barbara Liskov suggests in [9] a Java implementation using the state pattern of Gamma et Al. This proposal separates the type being implemented from the type used to implement it. Although this

[7] In the next examples we will use OASIS syntax although, as it has been showed in previous examples, there exist a direct correspondence with generalization in UML class diagrams.

implementation pattern provides a better modularity it only takes into account the trivial situation: one dynamic classification partition but no other classifications of the parent class at the same time. In this situation a more general solution is needed which can be extended to allow implementing all cases. Let us consider the next dynamic classification of the C class:

```
C1 where {atr < 10},
C2 where {atr >= 10} dynamic specialization of C;
```

For each child class belonging to the dynamic classification an inner class is written:

```
public class C
 {int atr;
 // subclass instances
 C1 c1; C2 c2;
 Public C (int v)
  {setAtr(v);}
 public void setAtr(int v)
  {atr=v; setDyn();}
 // migration engine
 private void setDyn()
  {if (atr<10)
   {c1=new C1(…); c2= null;
   else {c2=new C2(…);
   c1=null;}
  }
```

```
// C1 perspective
public C.C1 asC1()
 {if (c1!=null) return c1;
  else throw new
  NullPointerException();}
// C2 perspective
public C.C2 asC2()
 {if (c2!=null) return c2;
  else throw new
  NullPointerException();}}
private class C1
 {… //C1 is an inner class}
private class C2
 {// C2 is an inner class}
}
```

The child classes `C1` and `C2` have full visibility of `C` features or associations. The method `setDyn()` implements the corresponding child object creation. The methods `asC1()` and `asC2()` of `C` return the respective objects of the defined inner classes. This is necessary when we need to refer to emergent features or associations of child classes. An exception is triggered if the object does not belong to the child class and a request is made for child class features or associations. This exception is a `NullPointerException` what needs to be caught in the client object. The object creation of `C` instances follows the usual Java syntaxis: `C myObj = new C(...)`. The child classes features and associations of `myObj` object are available in `myObj.asC1()` and `myObj.asC2()`. Another implementation alternative is to specify child classes out of the parent class, solving features and association visibility by means of the *delegation* mechanism. The drawback in this case is the bidirectional communication needed: (1) child classes need to see inherited features and associations; and (2) the parent class needs to maintain references to the objects which are in the child classes, and any feature or association demanded at the parent class needs to be routed. We have chosen the inner class alternative because one way of communication is easily solved.

3.3 Static Classification (Complex)

When there are more than one static classification from the same parent class the situation is more complex because it is potentially possible to create an object in any case given by the possible species. An important matter is to know which is the most

frequent child class in where the objects should be created. Because the event new must be specified with concrete information (available at the lowest level) we have discarded the object creation at the parent class. It is impossible to design a good architecture which facilitates the object creation from any species. For this reason we have decided to choose one classification and we will refer to it as the *primary classification*. The idea is to implement an inheritance schema which involves the primary classification and which makes use of the *extends* Java language mechanism. For example, let us consider the following static classifications:

```
B1, B2 static specialization of C;
C1, C2 static specialization of C;
```

Where `B1` and `B2` are child classes of the primary classification. A first approximation in Java language would be the next one:

```
public abstract class C {...}
public class B1 extends C {...}
public class B2 extends C {...}
```

Then, the creation of objects is made as before at the leaf classes:

```
B1 objB1 = new B1{…};        B2 objB2 = new B2{…}
```

The next step is to complete the description of the class `C` including those child classes of non-primary classifications. The class `C` implementation adds the rest of child classes as inner classes:

```
public abstract class C
{public class C1 {…}
 public class C2 {…}…}
```

The object creation begins as `B1 objB1 = new B1(…);`. Once an object has been created then we need to specialize it in each of the non-primary classifications. In our example we would write `C.c1 objB1xC1 = objB1.new C1(…);`. That is, the object `objB1xC1` represents the specialization perspective in that non-primary classification and all the new or overwritten requested features or associations need to be routed to it. The programmer has the responsibility of assuring the atomicity of these two creations.

3.4 Dynamic Partitions (Complex)

Now let us see the situation in which there is more than one dynamic classification from the same parent class. In the next example, we have two dynamic classifications with the class `C` as parent class:

```
C1 where {atr<10},
C2 where {atr>=10} dynamic specialization of C;
D1,D2 dynamic specialization of C migration relation is
C = new().D1;  D1 = m2().D2; D2 = m3().D1 + m4().D1();
```

In this example, when the object has just been created it begins as a `D1` instance. When `m2()` occurs then the object migrates to the child class `D2`. Afterwards, occurrences of `m3()` or `m4()` produce the migration to the child class `D1`. When using

UML a state chart diagram need to be used to represent the migration relatioship. The simplified Java code for this case is:

```
public class C
 {
int atr;
  C1 c1; C2 c2;
  D1 d1; D2 d2;
 public C (int v)
  {atr=v; setDyn();
   d1 = new D1(…);}
 public void m2()
  {d2 = new D2(…);
   d1 = null;}
public void m3()
  {d1=new D1(…);d2=null;}
public void m4()
  {d1=new D1(…);d2=null;}
private void setDyn(){…}
public void setAtr(int v){…}
private class C1 {…}
private class C2 {…}
private class D1 {…}
private class D2 {…}
public C.C1 asC1() {…}
public C.C2 asC2() {…}
public C.D1 asD1() {…}
public C.D2 asD2() {…}
}
```

The implementation is similar to the simple situation, that is, in the dynamic classification child classes are implemented as inner classes of the parent class. The parent class implements the migration process.

4 Other Implementations

In this section we describe two other possible implementations using interfaces and design patterns. Choosing interfaces, in order to establish the semantics we could use Java Modeling Language (JML). JML [10] is a behavioral interface specification language that can be used to specify the behavior of Java modules. It combines the approaches of Eiffel and Larch, with some elements of the refinement calculus. Eiffel and Larch are well known, and the refinement calculus is a formalization of the stepwise refinement method of program construction. The required behavior of the program is specified as an abstract, possibly non-executable, program which is then refined by a series of correctness-preserving transformations into an efficient, executable program. JML allows including constraints when the interface is specified. However, when a class implements such an interface its contract forces it only to accomplish with the syntaxis, because the constraints imposed by JML are exclusively syntaxis constraints. Another approach would be using the Role Object [11] design pattern. This design pattern adapts an object to the different needs of the clients through transparently attached role objects, each one representing a role that the object has to play in that context of the client. The object manages its role set dynamically. By representing roles as individual objects, different contexts are kept separate and system configuration is simplified. Generalization from the point of view of this design pattern is based on the fact that the definition of a class changes according to the semantic demands on it. Therefore, this pattern is appropriated because in the generalization it is necessary to handle the available roles dynamically, thus they can be attached and removed on demand, that is, at run-time rather than statically at design-time.

5 Related Work

Although generalization is a key concept in the object-oriented approach, the UML specification only dedicates a few pages to its definition, leaving many aspects open to interpretation. Few works has been reported in order to give a more precise semantic for UML class generalization and we have not found references of works translating UML class generalization of analysis models into implementation. Some details are given in [12] and [13] about what a static and a dynamic classifications should be. However, many books and CASE tools claim that they establish or include code generation from UML models, and particularly considering class generalization. Unfortunately, after having a look at what they offer we usually find out that they consider only a simplified version of static classification, Furthermore, they do not distinguish more than one classification having the same parent class. Maybe the work by Gamma et Al. [14], and especially their state pattern is the more popular approach when there is some concern about translating dynamic classification into inheritance in a programming language. However, firstly, the state pattern is a design pattern, that is, useful in the design level (when an abstraction of the implementation must be established) and it is not comparable to the more abstract dynamic classification patterns offered in OASIS (they are patterns at the analysis level). Secondly, the state pattern does not establish the treatment of several orthogonal classifications with the same parent class, or other considerations like migration by means of events.

6 Conclusions

When considering only one static classification with the same parent class there would be no problems in translating UML class generalization into an object oriented programming code. However, classifications can also be dynamic and several classifications can have the same parent class. In addition, the translation encloses a number of considerations and decisions regarding the semantic associated to the classifications. This semantic is left open enough in UML, what makes necessary to take some more precise framework. In this work we have used OASIS generalization as an intended semantic for UML class generalization. When inheritance is seen only from an implementation perspective then those theoretical and conceptual features (such as behavior compatibility) are ignored and others are more relevant, such as reuse of coding, performance, etc. Most programmers see inheritance as a mechanism of incremental modification which allows programs to be extended or refined without changing the original code. The discussed implementation patterns in Java use *inner* classes and the *extend* mechanism. Before choosing the pattern for static classification we put the constructor method at the parent class level. Then we have looked for a way in which any object could be created (from the parent perspective) but we have not found an alternative solution due to the possible existence of several classifications from the same parent class. When we offer dynamic classification we increment the expressiveness although the structural clarity is reduced because it is not easy (neither formal) to deduce child class properties from the parent class. The

class generalization cases analyzed in this paper are not all the possible situations. Nevertheless, the solutions provided should be easy enough to extend to more specific modeling scenarios. Although OASIS is a formal language used to specify analysis models, from its origins there has been interest in using it as a support for industrial environments for software development. Thus, around OASIS three aspects have always been present : software development process, tool support, and automatic code generation from models oriented to model validation or to obtain final code[8].

References

1. Object Managment Group. OMG Unified Modeling Language Specification (v. 1.4), 2001
2. Taivalsaari A. *On the Notion of Inheritance*. ACM Comp. Surv., Vol. 28(3) (1996) 438-478
3. Al-Ahamad W. and Steegmans E. *Integrating Extension and Specialization Inheritance*. Journal of Object-Oriented Programming, December (2001)
4. Wegner P. and Zdonik S. *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like*. In Proc. of the 7th. European Conference on Object Oriented Programming, LNCS 322, ECOOP'88 (1988) 55-77
5. Letelier P., Ramos I., S nchez P. and Pastor O. *OASIS 3.0: A Formal Approach for the Object Oriented Conceptual Modeling*. Technical University of Valencia, ISBN 84-7721-663-0, Spain, http://www.dsic.upv.es/users/oom/books.html. (in Spanish) (1998)
6. Wieringa R., Jonge W. and Spruit P. *Using Dynamic Classes and Role Classes to Model Object Migration*. Theory and Practice of Object Systems, Vol. 1(1) (1995) 61-83
7. Letelier P., S nchez P., Troyano J. and Crespo Y., *Specialization in Conceptual Modeling: A rigurous use of Inheritance*. Actas del 3er Workshop Iberoamericano de Ingenier a de Requisitos y Ambientes Software (IDEAS), Canc n, M jico (in Spanish) (2000)
8. S nchez P., Letelier P. and Ramos I. *Animating Formal Specifications with Inheritance in a DL-Framework*. Requirements Eng. Journal, Vol.4, Springer-Verlag (2000) 198-209
9. Liskov B., Guttag J. Program Development in Java: Abstraction, Specification and Object-Oriented Design. Addison-Wesley (2001)
10. Leavens G., Rustan K., Leino M., Poll E., Ruby C. and Jacobs B. *JML: notations and tools supporting detailed design in Java*. In OOPSLA '00 Companion, Minneapolis, Minnesota, Copyright ACM (2000) 105-106
11. B umer D., Riehle D., Siberski W. and Wulf M. *Role Object*. In Pattern Languages of Program Design 4. Edited by Neil Harrison, et Al. Addison-Wesley, Chapter 2 (2000) 15-32
12. Martin J. and Odell J. *Object-Oriented Methods: A Foundation*. Prentice Hall (1998)

[8] Information about these lines of works at www.dsic.upv.es/users/oom.

13. Fowler M. and Kendall S. UML Destilled: Applying the Standard Object Modeling Language. Addison-Wesley (1997)
14. Gamma E., Helm R., Johnson R. and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, MA (1994)