Language support for Adaptive Object-Models using Metaclasses ⋆

Reza Razavi [a] Noury Bouraqadi [b] Joseph Yoder [c]
Jean-François Perrot [d] Ralph Johnson [c]

[a] *Software Engineering Competence Center - University of Luxembourg*
6, rue Richard Coudenhove-Kalergi - Luxembourg, L-1359- Luxembourg
reza.razavi@univ.lu

[b] *Ecole des Mines de Douai - Dépt. G.I.P*
941, rue Charles Bourseul - B.P. 838 - 59508 Douai Cédex - France
bouraqadi@ensm-douai.fr

[c] *Department of Computer Science*
University of Illinois at Urbana-Champaign - Urbana, IL 61801 - USA
yoder@refactory.com - johnson@cs.uiuc.edu

[d] *Laboratoire d'Informatique de Paris VI (LIP6)*
Université Pierre et Marie Curie - CNRS - Paris, 75252 - France
jean-francois.perrot@lip6.fr

Abstract

Adaptive Object Models are a sophisticated way of building object-oriented systems that let non-programmers customize the behavior of the system and that are most useful for businesses that are rapidly changing. Although systems based on an Adaptive Object Model are often much smaller than competitors, they can be difficult to build and to learn. We believe that the problems with Adaptive Object Models are due in part to a mismatch between their design and the languages that are used to build them. This paper describes how to avoid this mismatch by using implicit and explicit metaclasses.

Key words: Adpative Object Model (AOM), TypeObject Pattern, Meta-Object, Implicit Metaclasses, Explicit Metaclasses

⋆ The work communicated in this paper has been conducted while the first author doing his PhD at Laboratoire d'Informatique de Paris 6 (LIP6), Université Paris 6 - CNRS, Paris, France.

1 Introduction

Rapid change in business practice creates the need for software development approaches that permit rapid changes in software. The next generation of software systems must be sufficiently malleable and sensitive to business dynamics to allow businesses to adapt to the shifting operating environments. The Adaptive Object-Model Architectural Style [1] is born from the quest for such software by pioneering business organizations [2–4]. Recent academic researchers have documented the recurring design patterns of those systems [5–11].

Adaptive Object-Models (AOM) are object-oriented applications that use regular objects, i.e. instances, for representing metadata that describes the desired structure and behavior of the software in a given operating environment. For instance, the object-model, business rules [12] and roles relevant to a specific insurance or finance product can be specified at runtime by metadata. Business experts change the metadata using graphical tools to reflect domain changes in the software. The user-generated metadata is interpreted and this leads to "immediate", but controlled, effects on the system interpreting it. This is different from traditional configuration since this architectural style enables runtime and intuitive modification of the class hierarchy of the object-oriented program. Opening the system for intervention of "non-programmers" at runtime is one of the major assets of AOMs. Indeed, as it is observed by Bonnie A. Nardi [13], domain experts have the detailed task knowledge necessary for creating the knowledge-rich applications they want and the motivation to get their work done quickly, to a high standard of accuracy and completeness. AOMs are designed for empowering experts and make it possible:

- To develop and to change software quickly. AOM reduces time-to-market, by giving immediate feedback on what a new application looks like and how it works, and by allowing users experiment with new product types,
- To modify a software in accord with business experts, without calling programmers,
- To avoid shutting down the system in order to adapt it to new or local business needs,
- To reduce the volume of code that programmers should develop and maintain.

AOMs have the potential for providing businesses with highly competitive tools to cope with changes resulting from recurrent merges, alliances, acquisitions, etc. In short, AOMs correspond to an industrial reality, and have the potential to assist organizations in coping efficiently with their business evolution. They represent an important, long-term trend in software engineering. However, AOMs can require more effort to develop. Several reasons that have

been outlined in the literature are:

- An AOM can be hard to implement since it has more complex requirements, e.g. the need for producing, storing and interpreting metadata,
- Since an AOM leads often to a domain-specific language, therefore, as for any language, developers should provide programming tools such as debuggers, version control, and documentation tools,
- Developing database schemas and graphical user interfaces for AOM is harder since the specification of the underlying data changes at runtime,
- The architecture of an AOM can be harder to understand, document and maintain since there are two coexisting object systems; the interpreter written in the object-oriented programming language, and the object-model of the underlying business domain that is interpreted.

In this paper, we explore the use of reflection [14–16] for supporting AOM programming. More precisely, we focus on the use of metaclasses to support class adaptations made by business experts at runtime, while avoiding the mismatch between their design and the languages that are used to build them. So, computer professionals can maintain and evolve code changed by experts. We explored the use of Smalltalk-80 [17] implicit metaclases as an alternative to traditional techniques. Implicit metaclasses come with their drawbacks. Explicit metaclasses experimented with MetaclassTalk [18,19] proved more satisfactory.

The rest of this article is organized as follows. Section 2 describes the context of this research and states the research problem that we address. Section 3 explains how to address that problem using metaclasses. Section 4 presents the related work. Section 5 is devoted to conclusions and perspectives of this work.

2 Background on AOMs and the Problem Statement

The concept of AOM is born recently from the research aiming at discovering and documenting the design principles of a particular class of software with complex behavior that emerged from the industry. AOMs have been also called in the past "User Defined Product architecture" [2], "Active Object-Models" [8,7] and also "Dynamic Object Models" [9].

2.1 AOMs as Metadata interpreters

AOMs are software systems that *interpret an object-oriented representation of some business products and rules*. That representation is described as metadata [8] that specifies the latest object model of the business. Those specifications are stored in a database and loaded when necessary for building up the object model that represents the real business model that is interpreted to provide the desired behavior. So, the architectural style of AOMs emphasizes run-time adaptability by designing business software as a *metadata interpreter*.

Mentioning *metadata* is just saying that if something is going to vary in a predictable way, then you should store the description of the variation in a database so that operating the variation is easy. In other words, if something is going to change a lot, make it easy to change. The problem is that it can be hard to figure out which elements are going to change, and even if you know it then it can be hard to figure out how to describe the variations in your database. Code is powerful, and it can be hard to make your data as powerful as your code without making it as complicated. But when you are able to figure out how to do it right, metadata can be incredibly powerful, and can decrease your maintenance burden considerably. The most difficult part of developing an AOM is of course to figure out the model of the variations. This corresponds to a *metamodel* [20] since its instances represent the object models of the system.

Recent pattern mining effort [7,9,10] has allowed documenting the most fundamental underlying design patterns [21] such as (1) design for runtime-definition of classes, (2) design for runtime-definition of attributes, (3) design for runtime-definition of relationships, and (4) design for runtime-definition of behavior. In the terminology used in this paper, this activity is called *adaptation*, and the entity that results from this activity is also called an *adaptation*. The class that is extended by an adaptation is called an *adaptive class*. Each adaptive class models an evolving element of the changing domain, like Videotape in the example that follows, where Terminator is one of its possible adaptations.

The focus of this paper is only on design for the runtime-definition of a class by a domain expert.

2.2 The Classic AOM Technique for Adapting a Class

The problem of design for runtime class definition in AOMs is a case of the recurrent problem faced by developers of large systems, having to design a class (generically called Component) from which an unknown number of subclasses

should be derived. A standard solution to this problem is documented by the TypeObject pattern [5]. It consists in representing the unknown "subclasses" of Component by simple instances of a class generically called ComponentType. New "subclasses" can be created at run-time by instantiating ComponentType. Instances of these "subclasses" are then created as regular instances of Component with an explicit pointer to the instance of ComponentType that represents their subclass. In this setting, instances of ComponentType are interpreted as *types* for the ordinary instances of Component.

As a simple example of the feature we want to exploit, consider the *Videostore* example of the Type-Object pattern given by Johnson & Woolf [5]. In this case, the business objects are videotapes containing movies (exactly one movie in each tape), which are rented to customers. Terminator n° 20, n° 21, etc. are examples of videotapes of the movie called Terminator. Of course, the store will own and rent several videotapes of the same movie. End-users of the system will deal with existing videotapes, manage their rental to customers and conduct the bookkeeping. Maybe they will sometimes acquire new videotapes for existing films. Domain experts on the contrary will have to introduce new movies, with their title, rental price and rating.

The design problem is that the exact number of available movies can be known only at runtime (our client is not interested in a software that manages a predefined number of movies; it doesn t makes sense nowadays for this business). Applying TypeObject leads to a design with two classes: class Videotape as Component, and class Movie as ComponentType. Each instance of Videotape carries a metaobject (instance of Movie) which represents the movie and contains information shared by all Videotapes of the same title.

It should be noted that class Movie represents here the class model of the interpreter embedded in the AOM. Its instances (generated at run-time) constitute the metadata that represents the different videotapes.

In this example, Videotape should then be an adaptive class, whereas Customer remains an ordinary class (since our domain model does not contemplate its modification: one may suppose that it is fixed by other constraints, such as corresponding to an external database, etc). A usage scenario of this design would be something like:

```
|term term1 term2 joe|
term := Movie title: 'Terminator'.
term1 := Videotape movie: term.
term2 := Videotape movie: term.
joe := Customer new.
term1 rentTo: joe. etc...
```

where title: and movie: are creation methods of classes Movie and Videotape

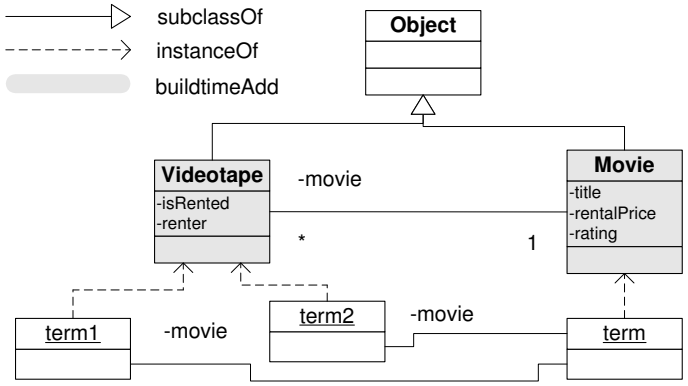

Fig. 1. Class & instance diagram of the Videostore example, with TypeObject design.

respectively. Instance term1 of Movie, representing Terminator, is the meta-object common to instances term1 and term2 of Videotape. Fig. 1 provides the diagram of this example. Classes created on software build-time (i.e. by computer professionals) are grayed.

It should be noted that domain experts execute this scenario or other presented in this paper using a GUI. The code described all along this paper corresponds to what happens behind the scene, from the programmers' point of view.

Here is the relevant fragment of code borrowed from [5], to be compared with the code resulting from approach presented in the following sections.

```
Movie class>>title: aString
   ^self new initTitle: aString

Object ()
   Videotape (movie isRented renter)
   Movie (title rentalPrice rating)

Movie>>newVideotape
   ^Videotape movie: self

Videotape>>initMovie: aMovie
    movie := aMovie

Videotape>>rentTo: aCustomer
    self checkNotRented.
    aCustomer addRental: self.
    self makeRentedTo: aCustomer
```

```
Videotape>>checkNotRented
    isRented ifTrue: [^self error]

Customer>>addRental: aVideotape
    rentals add: aVideotape.
    self chargeForRental: aVideotape rentalPrice

Videotape>>rentalPrice
    ^self movie rentalPrice

Videotape>>movie
    ^movie

Videotape class>>movie: aMovie
   ^self new initMovie: aMovie

Movie>>rentalPrice
    ^rentalPrice

Movie>>initTitle: aString
    title := aString
```

2.3 Business Objects and their MetaObjects: the Mismatch Problem

The following discussion hinges on the distinction between non-terminal and terminal objects (or, respectively, classes and non-classes). A terminal object is not a class; rather it is an instance of a class. The expression "non-terminal object" is equivalent with "class". As an abbreviation, it is often practical to transfer the qualification of instances to their classes: accordingly we shall speak of a *terminal class* to mean a class whose instances are terminal objects, as opposed to a metaclass, whose instances are classes [22].

The central problem in AOM design is the representation of evolving business objects. It follows that a regular set of classes, of which the business objects would be instances, won't be enough. The definition of a business objects must be assigned to specific meta-objects whose role is to represent information about the implementation and the interpretation of the object. As it is described in Dynamic Object Model design pattern (DOM) paper [9], to address this issue AOMs apply currently the TypeObject design technique, where "type" objects represent information about the implementation and the interpretation of associated business objects (which is the role of a meta-object).

As explained above, this pattern introduces two separate classes, known generically as Component and ComponentType. Class Component takes care of the fixed part, and class ComponentType deals with the variable part. Accordingly, a business object will be realized as an instance of Component (the "object") coupled with an instance of ComponentType (the "meta-object"), by means of a pointer. Our analysis is that such a business object will then have two metaobjects, (1) its class, of which it is instance, and (2) the associated component type (instance of ComponentType). For instance, a business object resulting from this design such as the *Spiderman videotape n° 20* will have as metaobjects class Videotape as well as an instance of Movie.

In our running example, Movie is an "ordinary" class, and therefore its instances are non-classes. However, according to TypeObject, each of these instances plays a class-like role for all instances of the class Videotape to which it is related (this role might be described as a "class complement"). This association is created by the initMovie: method in Videotape class, which is itself called by the creation method movie: in the metaclass "Videotape class". In other words, each terminal instance of Movie represents a subclass of Videotape, as was said earlier. However, such a subclass does not exist in reality, and the objects that are considered as being its instances, are in reality instances of Videotape (instances of Movie being terminal, it is impossible to instantiate them).

As an example of the undesirable consequences of this design, when programmers "inspect" the *Spiderman videotape n° 20*, they see that it is an instance of Videotape with an instance variable "type" that refers to a Movie. It would be more helpful for them to see that this videotape is an instance of Spiderman, which would be at the same time a subclass of Videotape and an instance of Movie.

The classic AOM solution thus leads to a conceptual and technical problem from the point of view of AOM programmers and maintainers. The origin of this problem is the mismatch between the semantics of TypeObject and that of the underlying object-oriented language, here Smalltalk-80, in their approach to subclassing. The consequences of this mismatch are well documented in the Dynamic Object-Model design pattern paper as *increased design complexity, increased runtime complexity*, and *new development tools*. The goal of this communication is to present an alternative approach that will avoid this mismatch.

3 Solution by Using Metaclasses

Our proposal is to use as meta-objects those objects that naturally play such a role, namely classes. Of course, this will entail some extensions to the traditional design of classes. This is precisely here that metaclasses [23,22,24] come into play.

Subsection 3.1 gives an overview of metaclass use. Subsections 3.2 and 3.3 describe and illustrate the use of respectively implicit and explicit metaclasses for implementing component types. Subsection 3.4 provides a summary of the approach and its results.

3.1 Overview of Metaclass use

Our design is strongly influenced by Smalltalk and by Pierre Cointe's ObjVlisp model [25,26]. In Smalltalk the idea that classes are indeed objects takes the following strong form: any class may be endowed with a set of instance variables and a dictionary of methods which gives it an individual behavior as an "ordinary" object. In normal practice, this facility is mainly used to define instance creation methods. We propose to make use of the same facility to endow a class X with all the structure and behavior needed to turn it into the metaobject of its instances. As for any object, the way to define those instance variables and methods is to write them in the class of which X is instance. Since X is a class, its class is a metaclass.

To proceed further we need to be more specific about the status of metaclasses. As is well-known, introducing metaclasses leads to a number of non-trivial difficulties for which we have to choose a solution. We shall consider (1) standard Smalltalk (2) an extension of Smalltalk called MetaclassTalk which is the latest development of a long line of research [19,18].

Standard Smalltalk [17] imposes very strict limitations on its metaclasses. With each class C a metaclass called "C class" is automatically associated, of which C is the only instance. "C class" can be edited to receive additional instance variables and methods and hence change the structure and behavior of C. If class B is a subclass of C, then its associated metaclass "B class" is automatically a subclass of metaclass "C class" inheriting the additional structure and behavior that was added to class C and which therefore applies also to B.

The solution *idea* is then to implement *ComponentTypes* as metaclasses. In this context, adding a component type corresponds to adding a new metaclass. Adding a new component means then instantiating the relevant metaclass. To

illustrate this idea, consider again our running example. We propose that *each movie* (call it M) should be represented as a subclass of Videotape, of which those *videotapes* that contain M will be instances. To this end we must endow these classes with the necessary (meta)behavior by means of their metaclasses. A simple way of realizing it in standard Smalltalk is to define the metabehavior in the metaclass "Videotape class", and to take advantage of the parallel inheritance of metaclasses.

The plain Smalltalk-based solution works fine, but since a metaclass can have only one instance the designer cannot reuse the same instance behavior with various meta-behaviors. Also, metaclasses are not treated as ordinary classes, so that there is no possibility of applying the same scheme to a metaclass in order to obtain multiple ontological levels.

In MetaclassTalk things are different. Metaclasses exist in an independent fashion and are created as such. When creating a class, its metaclass can be specified as well as its superclass. It is therefore possible to use both superclass specification for instance behavior, and metaclass specification for class behavior.

3.2 Use of Implicit Metaclasses

The first model of metaclasses that we propose to experiment is that of implicit meta-classes implemented by Smalltalk-80. This model can be sketched as follows:

The metaclass of each class is chosen (created) automatically by the implementing system. Such metaclasses are implicit: they are both anonymous and developers don't handle them directly. The system manages the metaclass inheritance hierarchy and makes it be parallel to the class inheritance hierarchy. Because of these parallel hierarchies, properties (i.e. structure and behavior) of a given class are automatically propagated to sub-classes.

The Smalltalk-80 model of implicit metaclasses supports specialization by means of a *framework* whose extension points are the Object class, the Class metaclass, and the Metaclass metaclass. Object is the root of the hierarchy of classes that describe the structure and behavior of non-classes. Class is the root of metaclasses that describe the structure and behavior of classes. Metaclass is the root of the *meta-metaclass hierarchy* that describe the structure and behavior of implicit metaclasses.

The typical case in Smalltalk is to use Metaclass as the default meta-metaclass. However, it is possible to use a specific meta-metaclass. Figure 2 below provides an example excerpted from VisualWorks version 5i.4 [27]. This system

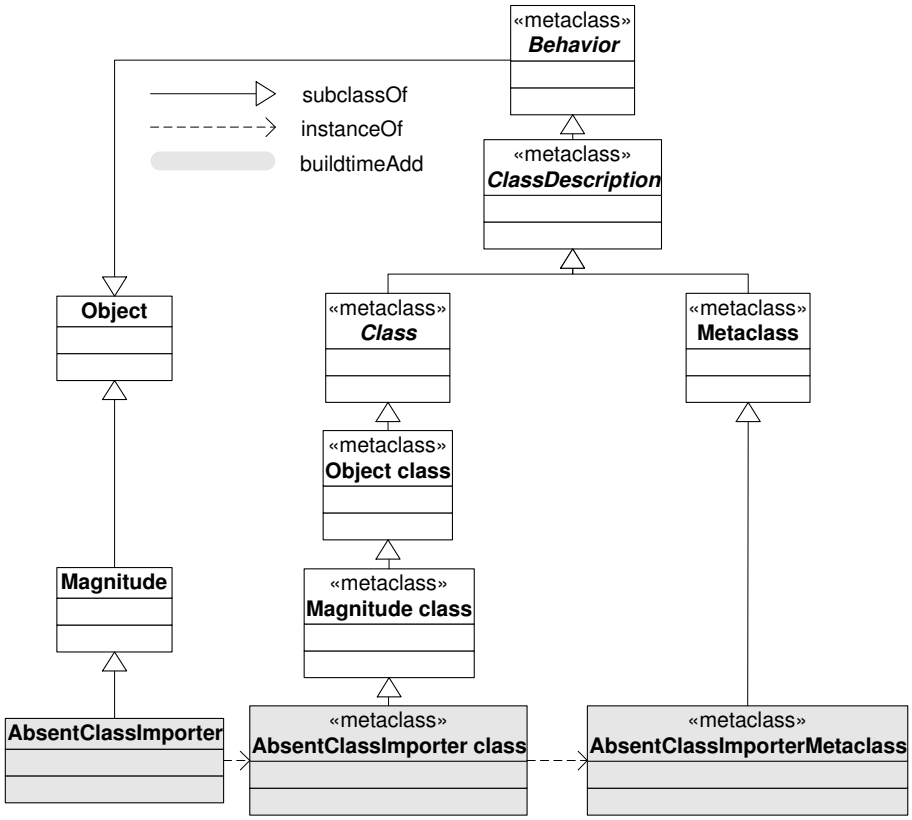

Fig. 2. Example of a meta-metaclass use in Smalltalk-80 (only main instantiation links are drawn).

enables loading a class whose superclass is absent from the system. It introduces new subclasses at the three levels: meta-metaclass AbsentClassImporterMetaclass, metaclass "AbsentClassImporter class" and class AbsentClassImporter.

For having a *ComponentType* created as an implicit metaclass, in the context of rules imposed by this model, programmers should subclass the "Object class" metaclass or one of its sub-metaclasses. Since implicit metaclasses are entirely managed by the Smalltalk system, a *Component* A corresponds in this model always to the sole instance of the *ComponentType* "A class". Definition of an adaptation C of the *Component* A inherits from A itself, while its metaclass "C class" inherits from "A class".

Illustration

Code portions below give the elements of the Smalltalk code corresponding to the implementation of the Videostore example by means of implicit metaclasses. Fig. 3 provides the corresponding class diagram. Classes and metaclasses built par computer professionals appear in gray while those created by

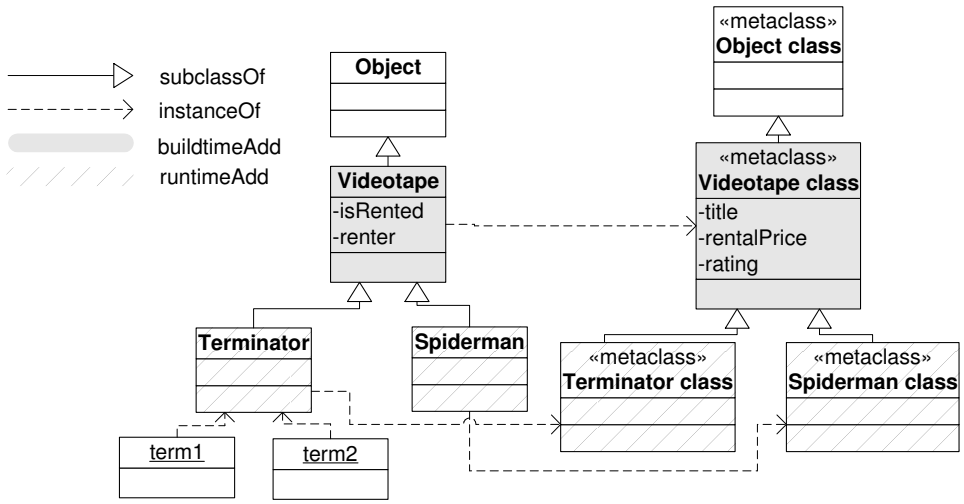

Fig. 3. Class & instance diagram of the Videostore example, with "Implicit Metaclasses" design.

domain experts are hatched.

We first have to define Videotape as a subclass of Object:

```
Object subclass: #Videotape
       instanceVariableNames: 'isRented renter '
       category: 'Videostore'
```

This creates also automatically the implicit metaclass "Videotape class" which is the *ComponentType* metaclass for videotapes. Here "Videotape class" will play the role of Movie in the TypeObject design above. We still need to add the instance variables of Movie, i.e., title, rentalPrice, and rating to "Videotape class". The resulting class hierarchy is as follows:

```
Object class ()
   Videotape class (title rentalPrice rating)
```

The creation method defined in this metaclass produces subclasses of Videotape whose title is received as argument:

```
Videotape class>>title: aString
   |sbcl|
   sbcl := self
       subclass: aString asSymbol
       instanceVariableNames: ''
       category: 'Videostore'.
   sbcl initTitle: aString.
   ^sbcl
```

Which requires the initialization method:

```
Videotape class>>initTitle: aString
    title := aString.
```

Now the access method:

```
Videotape class>>rentalPrice
    ^rentalPrice
```

is needed for

```
Videotape>>rentalPrice
    ^self class rentalPrice
```

The instance variable movie is no longer necessary. This link corresponds now to the instantiation link, managed by the Smalltalk virtual machine (the class message). This ensures better performances for AOMs.

```
Videotape>>movie
    ^self class
```

The remainder of the code is identical.

In the same spirit as the scenario above, the expert can now add new videotapes. The code that is executed behind the scene, since the expert uses a GUI, is as follows:

```
|term term1 term2 joe|
term := Videotape title: 'Terminator'.
```

so that after making several cassettes of this new movie:

```
term1 := term new.
term2 := term new.
```

they can be rented to customers:

```
joe := Customer new.
term1 rentTo: joe. etc...
```

In this setting, the movie Terminator is represented by a subclass of Videotape - the fact that this subclass might be named Terminator is of secondary importance. What is essential is that the metaclass "Terminator class" is automatically a subclass of "Videotape class". Seen as an object, class Terminator inherits the three instance variables defined in Videotape class, i.e. title, rentalPrice, and rating as well as the associated behavior. These instance variables are

set when the subclass is created, by means of the creation method Videotape class≫title: which calls the initializer initTitle:. In this way, class Terminator is fully equipped as a metaobject for the instances it generates, which represent individual cassettes of the movie Terminator. This setting is fully compatible with the object-oriented paradigm both from conceptual and technical points of view.

Now users will have to deal with instances of e.g. class Terminator (renting them to customers), with the creation of instances (buying new cassettes for the store), and with the creation of such classes (introducing new movies into the system). Note that creating a subclass at run-time does not require a different GUI than instantiating an existing class. The code we propose is easily packaged with a GUI which will completely hide the difference in implementation to both end-users and domain experts - but not, of course, to programmers!

The adaptations of Videotape like Spiderman and Terminator are now real subclasses. They can then be edited by programmers using their regular tools. The evolutions of the class hierarchy by experts can then be easily inspected and modified by professional programmers.

3.3 Use of Explicit Metaclasses

An alternative model to Smalltalk-80 implicit classes is that of "explicit" metaclasses. Object-oriented languages that implement this model allow programmers choosing explicitly the properties of their classes [23] by choosing its metaclass upon the creation of each class. This approach has been adopted by several systems, e.g. CLOS, Classtalk [28], and SOM. The prototype that we have developed here uses the MetaclassTalk system [18]. MetaclassTalk [1] is a reflective extension of Smalltalk-80. It results from a series of research on metaclasses and their use for defining class properties, starting with the ObjVlisp model and Classtalk system [28]. MetaclassTalk addresses the metaclass composition and compatibility issues [29]. Smalltalk virtual machine is used for runtime support. Its latest release, that we have used, uses the Squeak flavor.

Figure 4 below illustrates how to implement a specialization in the context of MetaclassTalk explicit metaclasses. The example provided here is a refactoring of the excerpt of VisualWorks provided in Figure 2.

MetaclassTalk provides the class Object and the metaclass StandardClass as extension points. Object is the root of the class hierarchy where properties of

[1] http://csl.ensm-douai.fr/MetaclassTalk

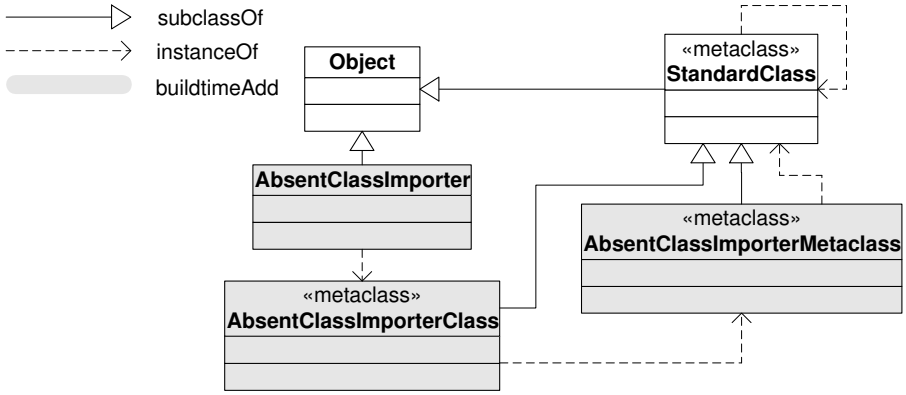

Fig. 4. Example of a specialization using explicit metaclasses.

non-classes are defined. StandardClass is the root of both the metaclass and the meta-metaclass hierarchies. Note that a metaclass is a meta-metaclass if its instances are also metaclasses, i.e. if they inherit (directly or not) from StandardClass. Adding a new abstraction systematically implies the extension of only one of the hierarchies described above. This extension is done *explicitly*. The programmer explicitly designates the superclass of the new abstraction. The programmer also explicitly chooses the type of the created class, by choosing the metaclass of the new class. Support for adapting classes relies then on the metaclass hierarchy.

In this context, to have their component types created as explicit metaclasses, programmers should subclass the StandardClass metaclass or one of its sub-metaclasses. As for the component class, it can be created with any appropriate superclass, e.g., Object. The choice of its metaclass is also only driven by the application needs and can be any metaclass in the system that provides the right behavior.

Illustration

Code portions below give the elements of the MetaclassTalk code corresponding to our running example.

We simply turn the implicit metaclass "Videotape class" of the design with implicit metaclasses into an explicit, free-standing metaclass which we naturally call Movie. Following normal practice in MetaclassTalk, as explained above, Movie is created as an instance and subclass of StandardClass (see Fig. 5).

```
StandardClass subclass: #Movie
    instanceVariableNames: 'title rentalPrice rating'
```

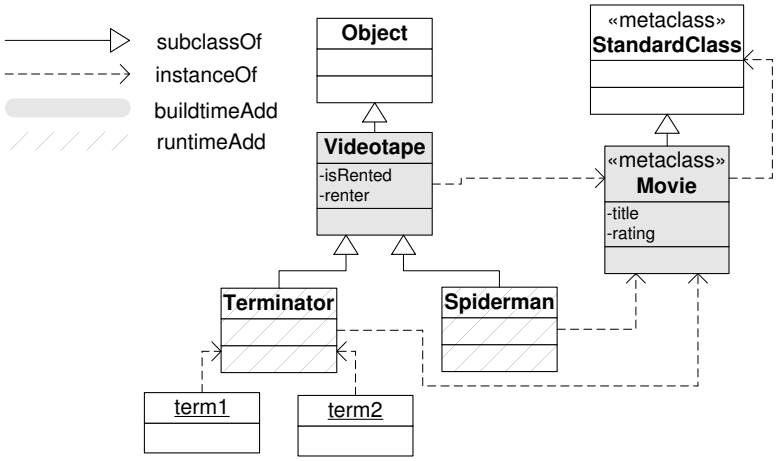

Fig. 5. Class & instance diagram of the Videostore example, with "Explicit Metaclasses" design.

```
    category: 'Videostore1'
    metaclass: StandardClass
```

Accordingly, class Videotape now appears as an instance of metaclass Movie:

```
Object subclass: #Videotape
    instanceVariableNames: 'isRented renter '
    category: 'Videostore1'
    metaclass: Movie
```

The resulting class hierarchy is as follows:

```
Object ()
   StandardClass (...)
       Movie (title rentalPrice rating)
           Videotape (isRented renter)
```

The creation method title: that was previously defined in the metaclass "Videotape class" is now the responsibility of Movie. Note that a typical receiver for this method will be class Videotape.

```
Movie>>title: aString
     |sbcl|
     sbcl := self subclass: aString asSymbol
        instanceVariableNames: ''
        category: 'Videostore1'
        metaclass: self class.
     sbcl initTitle: aString.
     ^sbcl
```

Same transfer for the other methods that were previously defined in "Videotape class":

```
Movie>>rentalPrice
    ^rentalPrice

Movie>>initTitle: aString
    title := aString.
```

The remainder of the code is identical, and the scenario above can be repeated without any change.

The decisive advantage of explicit over implicit metaclasses is the separation of the inheritance and instanciation hierarchies of class Videotape and of metaclass Movie. This allows for arbitrarily complex designs, using standard techniques (e.g. reuse) both for Videotape and for Movie.

This allows also for the construction of several levels of metaclasses, corresponding to the "nested type objects" of Johnson & Woolf. Fig. 6 below sketches the design for one of the extensions they add to the Videostore example, where *movies* may belong to different *movie categories*. In our design, MovieCategory is a meta-metaclass of which metaclass Movie is an instance. Individual *categories* appear as metaclasses that are instances of MovieCategory and subclasses of Movie. Class Videotape remains the same (instance Movie of and subclass of Object). Individual classes representing movies appear now as instances of the various categories (meta-classes) and remain subclasses of Videotape. In the example sketched in Fig. 6, there are two *movie categories* (named First and Second,) and two *movies*, Terminator belonging to category First and Spiderman to category Second. Only two videocassettes are available, both of Terminator. Note that the structural pattern of MovieCategory/Movie/First/Second is identical to that of Movie/Videotape/-Terminator/Spiderman, so that this design is actually simpler than it seems at first sight.

3.4 Discussion

The work communicated here is part of a larger-scale study, of which Razavi's doctoral dissertation [30] was the first version. It aims at designing a framework for AOMs (called in [30] DYCRA [2] for DYnamic Class Refinement Architecture). On the basis of the observations communicated here we plan to improve DYCRA by the systematic use of metaclasses (e.g. metaclass com-

[2] For more information please point to the URL: http://www-poleia.lip6.fr/∼razavi/Dyctalk/.

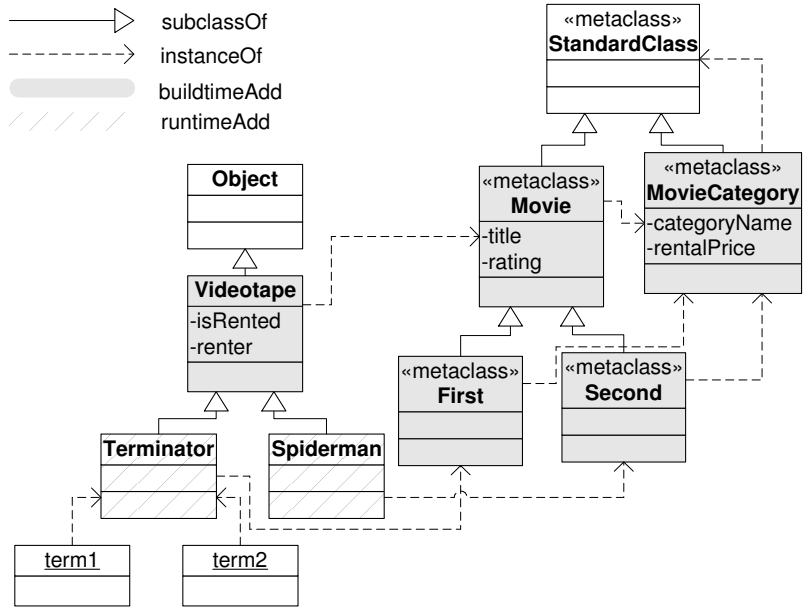

Fig. 6. Class & instance diagram of the extended Videostore example, illustrating the possibilities of the "Explicit Metaclasses" design.

position in the sense of Bouraqadi [19], Ducasse et al. [31]). The following remarks are set in this perspective.

One of the key issues is that AOMs have to undergo phases of refactoring where programmers will have to deal with a system on which experts have made a number of adaptations. If the architecture of the adapted code is complicated, then programmers will have a hard time doing their refactoring job. They will require specific tools and adequate training - and a different training for each new application.

This is why language-level support is essential, language being the common ground for all programmers (generic tools, etc). In Section 2 we pinpointed the mismatch problem due to an *ad hoc* architecture made necessary by the absence of reflective features, and in section 3 we showed how metaclasses can be used to solve this problem. Note that our design ensures that the "dynamic" classes that are created at run-time inherit all their methods from their "static" superclasses, so that the Smalltalk compiler is actually never called.

With our technique, the code remains indeed structured as a set of classes (as opposed to classes mixed with terminal instances in a design based on Type Object) programmers may use any engineering technique, e.g., editing adaptations using their regular editing tools, refactoring, even embedding a micro-workflow architecture [32], etc. Moreover, the code is indeed simpler, since a part of it disappears (the part of an AOM that represents the inter-

preter of the meta-level). The programming language itself is now called to play this role. Programmers of AOMs can then concentrate on modeling the business domain.

With implicit metaclasses (standard Smalltalk, section 3.2), metaclass reuse is limited. This becomes a challenging issue in designing reusable, language-level support for adaptation. That is, if we ant to empower AOM programmers with predefined support for implementing *ComponentTypes* as metaclasses. In this case, programmers should be able to develop their application-specific *ComponentTypes* by extending and reusing existing default metaclasses. For instance, the metabehavior associated to the metaclass "Videotape class" could be provided by the framework, and not implemented by AOM programmers.

As we saw in section 3.3, explicit metaclasses (with MetaclassTalk) eliminate this limitation. Infinite class hierarchies are indeed one of the AOMs fundamental features. MetaclassTalk appears therefore as a good candidate for an implementation language suitable for AOM design.

However, there is a price to pay. Choosing classes instead of arbitrary objects to represent meta-objects involves some limitations on the programmer's freedom. Instances belonging to two different classes will not be able to share the same meta-object. To follow Johnson & Woolf once more, a *Videodisk* and a *Videotape* will not share exactly the same Movie meta-object, since they belong to different classes - but they do not share the same rental price either. Our technique will lead to creating two classes with the same metaclass, e.g. TapeTerminator and DVDTerminator, and in order to express that these two classes are realizations of the same film it will be necessary to provide a supplementary ontological level in the same style as MovieCategory.

Also, changing the meta-object of a business object now involves changing its class, which is a risky operation. This is an important design issue. The gist of our approach is precisely to rely on language mechanisms, and paying the price for it. If the verifications and restrictions that go with it are undesirable, then our method is not the right one!

4 Related Work

Independently from AOMs, tool support for dynamic class specialization has been subject to extensive research since several decades in the field of object-oriented languages. Such a design has been introduced with Smalltalk-76. Extensions to that solution have also been subject to extensive research [25,26,33] [29,18,24]. What is different in the case of AOMs is the need for two different, but compatible, mechanisms for specialization. One is dedicated to program-

mers and the other to business experts.

Another comparable work is that around UML Virtual Machines [34], as well as executable UML models with the UML diagrams being supported by Action Semantics. There are two major issues with these approaches:

(1) the underlying solutions remain properties of private companies and have not been sufficiently documented as standard and reusable solutions; and
(2) they promote programming by UML-based languages, which is far from having gained popularity within the non-programmer business experts. Empirical research in the field of end-user programming shows that business experts are more likely to adopt domain-related solutions [13] and not an object modeling language designed for programmers. AOMs already implement such solutions. We believe that the applicability of UML profiles for building comparable solutions deserves more investigation.

AOMs are comparable with Domain Oriented Programming (DOP) languages, described by Dave Thomas & Brian Berry in [35]. Both DOP languages and AOMs embed a DSL (Domain-Specific Language). They are also designed "to allow knowledgeable end users to succinctly express their needs in the form of an application computation" [35].

Although the Squeak system [36] is not fundamentally designed for building AOMs, it addresses analogous problems since it provides the plain programming IDE as well as the EToy [37] interface. However, the pseudo-compatibility between classes and adaptations (Player class and its subclasses created implicitly during scripting by non-programmers using EToy) is achieved in an *ad hoc* manner.

Finally, this research has been influenced by work on metatool design for facilitating the creation of development tools that implement the "double-metamodeling" approach advocated by the METAGEN group [38]. In this system, one of the metamodels corresponds to a domain-related language for specifying the requirements and the other one corresponds to a technology-related language for expressing the implementation of these specifications. The process of moving from specification models to implementation ones is semi-automated, largely due to model transformation techniques, where NéOpus [39] is used for expressing transformation rules. This experimental work, going on since the early nineties, is closely linked with the more recent Model-Driven Architecture (MDA).

5 Conclusions and Perspectives

AOMs allow building business applications through the collaboration of programmers and business experts. This approach has been chosen by pioneer business organizations as a means to cope rapidly and cost-effectively with the need to adapting their critical software to the business dynamics. Programmers manage of course all technical aspects of the development process. Part of their responsibility is to empower business experts to specialize the "default" class inheritance hierarchy of the system, by dynamically adding new object types using domain-related constructs.

The current approach for creating AOMs (e.g. as documented by the DOM pattern) relies on a non-reflexive programming style. This approach leads to a series of issues that make the creation and maintenance of AOMs hard and costly. In order to help building AOMs while avoiding issues of existing solution, we explored in this paper the use of metaclasses. On the basis of this study, we can conclude that languages offering explicit metaclasses provide better support for adapting classes. MetaclassTalk is an example of such language. Languages that implement implicit metaclasses, like Smalltalk-80, only address part of adaptation support requirements. Indeed, they lead to a series of issues, like the loss of natural inheritance as well as undesirable propagation to a whole inheritance hierarchy of the choice of the metaclass.

Ledoux and Cointe observe that metaclasses are reifications that serve to vary the default semantics of classes [23], and suggest considering each such variation as a class property. Our proposal can then be summarized as "*adaptability should be treated as a class property*".

A last point to discuss is the relative importance of the mismatch problem that is the subject of this paper from the general point of view of AOM design. This mismatch is certainly not the only reason for the difficulties with AOMs. For instance, suppose you want to add a feature to a system. It can be added by the expert using only the tools that the programmer gave him, or it can be added by the programmer herself. The programmer can (and does) use the expert's tools, but the expert cannot use the programmer's tools.

The problem this poses for the expert is that sometimes he wants to do something but can't, and has to ask the programmer for help. The programmer can either build a new tool to let the expert to do it (or change an existing tool) or can just add the feature. Often the programmer does a little of one and a little of the other.

The problem this poses for the programmer is that there are several ways to reach her goal. Suppose that the expert can make a change by finding a set of instances and changing all of them, or the programmer can make a tool

that does it all at once. Suppose the expert can implement a procedure by making a complex workflow and reuse it by copying it and editing it, or the programmer can implement it by adding a new primitive to the system. There is no easy answer to the question "which is better ?". An AOM requires the programmer to be able to think of the system on several levels at once, and most programmers are not educated to do this well.

So, which problems are most important? If mismatch with language features is most important then our framework Dycra should make it measurably easier to build AOMs. But if mismatch is a secondary issue then it might not. We can not figure it out by hard thinking! The only way to tell whether mismatch is the main problem is to try to eliminate it and see what happens. This is the issue that we have addressed in this paper. Now we must experiment with the large-scale framework to figure out its real impact. We plan such experimentation in the course of a new project that aims at providing tool support for building families of adaptive and personalized solutions in the context of Ambient Intelligence [40].

Acknowledgments

The authors gratefully acknowledge the support from the Software Architecture Group (SAG) at UIUC and from the Metafor project at LIP6. This research benefited from a UIUC-CNRS exchange program, directed by Gul Agha and J.-P. Briot.

References

[1] J. W. Yoder, R. Johnson, The adaptive object-model architectural style, in: 3rd IEEE/IFIP Conference on Software Architecture (WICSA3). IFIP Conference Proceedings 224. Jan Bosch, W. Morven Gentleman, Christine Hofmeister, Juha Kuusela (Eds.), Kluwer, 2002, pp. 3–27.

[2] R. Johnson, J. Oakes, The user-defined product framework, (unpublished document).
URL `http://st.cs.uiuc.edu/pub/papers/frameworks/udp`

[3] F. Anderson, R. Johnson, The objectiva telephone billing system, in: MetaData Pattern Mining Workshop), 1998.

[4] M. Tilman, M. Devos, A reflective and repository-based framework. implementing application frameworks, in: Implementing Application Frameworks (M.E. Fayad, D. C. Schmidt, R. E; Johnson ed.), Addison-Wesley, 1999, pp. 29–64.

[5] R. Johnson, B. Woolf, Type object, in: Pattern Languages of Program Design 3, Robert Martin, Dirk Riehle, and Frank Buschmann, eds., Addison-Wesley, 1997, pp. 47–66.

[6] R. Johnson, Dynamic object model, (unpublished document).
URL http://st-www.cs.uiuc.edu/users/johnson/DOM.html

[7] J. W. Yoder, B. Foote, D. Riehle, M. Tilman, Metadata and active object-models, in: Workshop Results Submission OOPSLA'98 Addendum, 1998.

[8] B. Foote, J. Yoder, Metadata and active object-models, in: Proceedings of Plop98. Technical Report wucs-98-25, Washington University Department of Computer Science, 1998.

[9] D. Riehle, M. Tilman, R. Johnson, Dynamic object model, in: Proceedings of the 2000 Conference on Pattern Languages of Programming (PLoP 2000). Technical Report number WUCS-00-29, Washington University Department of Computer Science, 2000.

[10] J. W. Yoder, R. Razavi, Metadata and adaptive object-models, in: ECOOP'2000 Workshop Reader; Lecture Notes in Computer Science, vol. no. 1964, Springer Verlag, 2000.

[11] J. W. Yoder, F. Balaguer, R. Johnson, Architecture and design of adaptive object-models, SIGPLAN Not. 36 (12) (2001) 50–60.
URL http://doi.acm.org/10.1145/583960.583966

[12] A. Arsanjani, Rule object: A pattern language for pluggable and adaptive business rule construction, in: Proceedings of PLoP2000. Technical Report wucs-00-29, Washington University Department of Computer Science, 2000.

[13] B. A. Nardi, A Small Matter of Programming: Perspectives on End User Computing, MIT Press, 1993.

[14] B. C. Smith, Reflection and semantics in lisp, in: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press, 1984, pp. 23–35.

[15] B. Foote, Objects, reflection, and open languages, in: ECOOP'92 Workshop on Object-Oriented Reflection and Metalevel Architectures, 1992.

[16] B. Foote, R. E. Johnson, Reflective facilities in smalltalk-80, in: Conference proceedings on Object-oriented programming systems, languages and applications, ACM Press, 1989, pp. 327–335.
URL http://doi.acm.org/10.1145/74877.74911

[17] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

[18] N. Bouraqadi, T. Ledoux, Aspect-Oriented Software Development, Addison-Wesley, Boston, 2005, Ch. 12 – Supporting AOP Using Reflection, pp. 261–282.

[19] N. Bouraqadi, Safe metaclass composition using mixin-based inheritance, Journal of Computer Languages and Structures 30 (1-2) (2004) 49–61, special issue: Smalltalk Language.

[20] N. Revault, J. W. Yoder, Adaptive object-models and metamodeling techniques, in: Ecoop 2001 Workshop Reader. kos Frohner (ed), LNCS, Springer-Verlag, 2001.

[21] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Addison-Wesley Longman Publishing Co., Inc., 1995.

[22] J. F. Perrot, Objets, classes, hritage : dfinitions, in: In R. Ducournau, J. Euzenat, G. Masini, and A. Napoli, editors, Langages et Modles Objets: Etats des recherches et perspectives, chapter 1, INRIA - Collection Didactique, 1998, pp. 3–31.

[23] T. Ledoux, P. Cointe, Explicit metaclasses as a tool for improving the design of class libraries, in: In Proceedings of ISOTAS'96 - JSSST-JAIST, Springer-Verlag, 1996.

[24] I. Forman, S. Danforth, Putting Metaclasses to Work, Addison-Wesley, 1999.

[25] P. Cointe, Metaclasses are first class: The objvlisp model, in: Conference proceedings on Object-oriented programming systems, languages and applications, ACM Press, 1987, pp. 156–162.
URL `http://doi.acm.org/10.1145/38765.38822`

[26] P. Cointe, The objvlisp kernel: a reflective lisp architecture to define a uniform object-oriented system, in: P. Maes, D. Nardi (Eds.), Workshop on Meta-Level Architecture and Reflection, North Holland Publishing Company, Amsterdam, New York, Oxford, 1988, pp. 155–176.

[27] E. Miranda, Meta-programming in a flexible component architecture, in: Metadata and Dynamic Object-Model Pattern Mining Workshop OOPSLA '98, 1988.

[28] M. H. Ibrahim, Reflection and metalevel architectures in object-oriented programming (workshop session), in: Proceedings of the European conference on Object-oriented programming addendum : systems, languages, and applications, ACM Press, 1991, pp. 73–80.
URL `http://doi.acm.org/10.1145/319016.319050`

[29] N. Bouraqadi, T. Ledoux, F. Rivard, Safe metaclass programming, in: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press, 1998, pp. 84–96.
URL `http://doi.acm.org/10.1145/286936.286949`

[30] R. Razavi, Outils pour les langages d'experts — adaptation, refactoring et rflexivit, Thse de doctorat, Universit Pierre et Marie Curie (Paris 6), LIP6, Paris, France (Nov. 2001).

URL
http://www-ftp.lip6.fr/ftp/lip6/reports/2002/lip6.2002.014.pdf

[31] S. Ducasse, N. Schrli, R. Wuyts, Uniform and safe metaclass composition, in: Proceedings of the ESUG Research Track. Also published in a special issue of the Elsevier international journal "Computer Languages, Systems and Structures", 2004, to appear in 2005.

[32] D. A. Manolescu, Workflow enactment with continuation and future objects, in: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press, 2002, pp. 40–51.

[33] J. P. Briot, P. Cointe, A uniform model for object-oriented langauges using the class abstraction, in: In Proceedings of the Tenth Internationl Joint Conference on Artificial Intelligence (IJCAI'87), Vol. 1, 1987, pp. 40–43.

[34] D. Riehle, S. Fraleigh, D. Bucka-Lassen, N. Omorogbe, The architecture of a uml virtual machine, SIGPLAN Not. 36 (11) (2001) 327–341.
URL http://doi.acm.org/10.1145/504311.504306

[35] D. Thomas, B. M. Barry, Model driven development: the case for domain oriented programming, in: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press, 2003, pp. 2–7.
URL http://doi.acm.org/10.1145/949344.949346

[36] M. Gudizal, K. Rose, Squeak - Open Personal Computing and Multimedia, Prentice Hall, 1999.

[37] B. J. Allen-Conn, K. Rose, Powerful ideas in the classroom: using squeak to enhance math and science learning, Comput. Entertain. 2 (1) (2004) 16–16.
URL http://doi.acm.org/10.1145/973801.973827

[38] N. Revault, H. A. Sahraoui, G. Blain, J. F. Perrot, A metamodeling technique: The metagen system, in: Proceedings of TOOLS 16, 1995.

[39] F. Pachet, J. F. Perrot, Rule firing with metarules, in: Software Engineering and Knowledge Engineering - SEKE '94, Jurmala, Lettonie, Knowledge System Institute, 1994, pp. 322–329.

[40] I. A. Group, Ambient intelligence: from vision to reality - for participation in society & business (2003).
URL http://www.cordis.lu/ist/istag-reports.htm