# Dynamic Attribute Addition through the use of Metaclasses

Kopanitsanos Antonis

Department of Informatics and Telecommunications,
University of Athens, Panepistimioupolis, Ilissia
GR15778, Athens, Greece
antonis@di.uoa.gr

**Abstract.** Class evolution is a key issue in object oriented systems evolution. In particular much research has focused in topics raised by class evolution either through coercion or versioning. In this paper dynamic attribute addition in terms of versioning is addressed. Dynamic attribute addition in this context concerns modifications of variables of objects and classes at runtime. A candidate mechanism based on the notion of metaclasses is proposed to serve for this purpose. A case study of the use of the proposed mechanism is presented. Finally an evaluation of the benefits and the drawbacks that emerge from its use is established.

## 1 Introduction

Object oriented models are believed to better and more naturally reflect the behaviour and organization of real world phenomena, incorporating more of the real world semantics. But the real world itself may change and new requirements may arise for an object-oriented system. So software developers should be able to perform a large number of changes to an already existing system so that the system may continue to suit the evolving user's needs.

The relevant process commonly known as class evolution addresses issues, like behavioural and structural evolution. Behavioural evolution deals with the ability to change the behaviour of objects. Structural evolution is concerned with reclassification of existing objects and altering of objects attributes. The latter is the special problem we focus our interest on, in this paper. In particular we are concerned with the ability to change dynamically a class's association with properties even if this class is already populated with instances. In this case special care must be taken with these instances, which they should always comply with the definition of their class even if this class has a newer version.

According to the Smalktalk meta-programming model (Goldberg and Robson, 1983) every class is itself an object and as such an instance of a metaclass. So a metaclass is a class of a class ie. the objects of this class can themselves act as classes.
The notion of metaclasses is not directly supported by popular object oriented languages like, C++ and Java. Nevertheless it should be examined as a candidate mecha-

nism to provide a solution to the problem of attribute addition and removal at run time.
In this paper we present a model based on the notion of metaclasses as a solution to the aforementioned problem.
The rest of the paper is organized as follows. In section 2 we present issues concerning class evolution and metaclasses. In section 3 we describe the example case we deal with in this paper. In section 4 using the example case we describe the problems that can be experienced if current languages are used. In section 5 we present the proposed approach and explain how it can support dynamic attribute addition and removal. Finally in section 5 advantages and disadvantages of the proposed approach are presented together with future work.

## 2 Class Evolution

As mentioned before object oriented models successfully describe real world phenomena. Objects combine data together with their interpretation. But the real world itself may change and new requirements for information, or functionality to manipulate information in the application may arise. So, software developers, working with an object oriented system should be able to modify classes or the class hierarchy so that they fully suit their needs.

According to [1] we can divide the changes one system may support into the following categories:

1) Changes concerning instance variables

| |
|---|
| Add a variable |
| Remove a variable |
| Rename a variable |
| Redefine the type of a variable |
| Change the inheritance origin |
| Change the default value |
| Modify other kinds of variables |

2) Changes concerning methods

| |
|---|
| Add a method |
| Remove a method |
| Rename a method |
| Redefine the signature |
| Change the code |
| Change the inheritance origin |

3) Changes concerning classes

| |
|---|
| Add a class |
| Remove a class |
| Rename a class |
| Modify other class properties |

4) Changes concerning inheritance links

| |
|---|
| Add a superclass to a class |
| Remove a superclass |
| Change superclass precedence. |

5) Changes concerning the associations of a class

| |
|---|
| Add a new association |
| Remove an existing association |
| Change the multiplicity of an existing association |

The possible changes shown in the above tables may have different impact for the underlying system. In particular the changes we focus our interest on in this paper may affect

- Clients of the system, the application programs.
- The objects, which should always comply with the definition of their class, even if this class has a newer version.

With respect to this problem there are two different approaches, the coercion and the versioning approach.

The coercion approach regards all modifications as corrective and obsoletes former specifications accordingly. All instances of a modified class must be updated (coerced) according to the new definition, i.e. objects must always comply with the most recent specification of their class. But as all objects are converted, the coercion approach may imply information loss. Moreover, all clients of the system must also be modified and recompiled to be in accordance with the most recent version.

The versioning approach does not regard change as corrective, so former class or class hierarchy specifications are not obsolete upon change, but remain valid in coexistence with newer specifications. Objects must be properly enhanced, so that they may consistently behave under any class version. An important impact is that existing clients of a system may be unaffected by the change, continuing to regard the system in the same way as before. This may be useful for example when it is expensive or time consuming to modify and recompile application programs.

As we know objects are created as instances of classes. According to usual object-oriented approaches after their creation they cannot change association with properties because that is determined at the time of instantiation by the initial specification of their class. Nevertheless because of class evolution they have their structure altered by attributes added or removed dynamically and they need to be provided with extra storage space in order to be accordance with the newer version of their class. At this point we should mention that special care must taken if the system provides persistence for the data.

A meta-class is a means to define the common properties of a set of classes. A meta-class is a “package” of properties. The assembly of properties in a meta-class package is sometimes called a meta-class structure. Creating an object with the properties defined in a meta-class is called instantiation of the meta-class. After that instantiation we obtain a new object, called a class, but now it is called also a meta-instance which has the properties packaged in the meta-class.

As mentioned before popular object-oriented languages like C++ and Java do not support directly this mechanism. However it can be implemented using the notion of static data they both provide. We come now to the description of the example case we deal with in this paper.

## 3 Example Case Description

Suppose we have a system that contains among other classes, the classes AdultPerson and IdentityCard the specification of which is presented below.

```
public class AdultPerson {
   int height;
   String surname;
   IdentityCard iD ;
   //methods
}

public class IdentityCard{
   int codeNumber;
   String dateOfIssue ;
   //methods
}
```

Further suppose that the extent of those classes (instances) exists. But due to evolving user's needs a new variable is required to be added to each one of them. More specifically we need to add a variable of String type to the `AdultPerson` class having the name `residence`, and also a String type variable having the name `authority` to the `IdentityCard` class.

## 4 An Approach based on Inheritance

If current languages like C++ and Java were used, the incorporation of a new variable in a class could have been achieved using a well known object oriented technique called inheritance.

In our example the derived subclasses could have the following specifications.

```
class EnhancedAdultPerson extends AdultPerson{
    String residence;
    EnhancedIdentityCard eID ;
    // methods
}

class EnhancedIdentityCard extends IdentityCard{
    String authority;
    //methods
}
```

After the insertion of the two subclasses, existing instances of classes `AdultPerson` and `IdentityCard` should be migrated to classes `EnhancedAdultPerson` and `EnhancedIdentityCard` accordingly. Moreover the reference, that every `AdultPerson` object holds, to its corresponding `IdentityCard` object should be replaced with a reference to an `EnhancedIdentityCard` object.

The use of inheritance either single or multiple leads to a number of well known problems including name collisions, homonymous attributes and common ancestor dilemma. Other problems that can be experienced particularly in our example are listed below:

a) Creation of complex hierarchies which are difficult to handle, if requirements for attribute addition are often.

b) Class explosion if for each attribute we want to add we have to insert to the system a new class.

c) Difficulties in attribute removal.

So we need a new approach in order to achieve attribute addition and removal in general, and in a dynamic fashion in particular.

## 5 Proposed Approach

We begin the description of the proposed approach by introducing a new concept concerning the presentation of attributes. Each one of these attributes is modeled as an instance of class Data the specification of which is presented below.

```
public class Data{
      String variableType ;
      String variableName ;
}
```

As we can see for each one of the attributes its name and type are stored (using a String format) in the instance variables of the class Data. We could also have stored information concerning modifiers like protected or private, determining the access to the attributes being modeled but we decided to let it as part of our future work because its implementation requires extra support from the underlying software system. So we let all attributes be public.

According to the proposed approach the system's class hierarchy should have the format presented in figure1. As we can see at the top of the hierarchy is placed an abstract class called metaclass. All the other classes of the system inherit from that class.

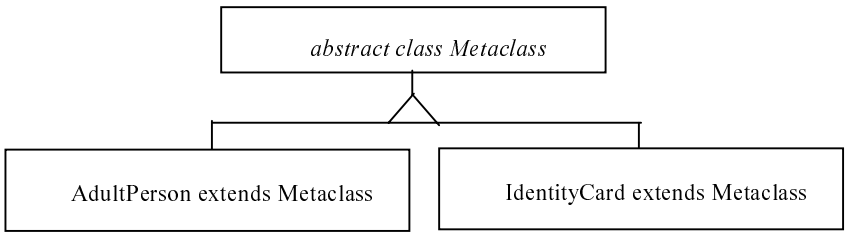



**Fig. 1.** Proposed Class hierarchy.

The structure of the abstract class Metaclass is as follows:

```
abstract class Metaclass {
      static Dictionary variableMap;
      Dictionary variableValue;
}
```

Because the variableMap Dictionary is declared as static there is only one copy of it for all the concrete subclasses of Metaclass, so it can be used as a repository storing information about the attributes contained in these subclasses. According to our approach the initial information to be stored in the static variableMap Dictionary seems like the one presented in the following table:

**Table 1.** variableMap Dictionary.

| "AdultPerson" | | |
|---|---|---|
| data1 | data2 | data3 |
| variableType:int<br>variableName:height | variableType:String<br>variableName:Surname | variableType:IdentityCard<br>variableName:iD |
| "IdentityCard" | | |
| data1 | data2 | |
| variableType:int<br>variablName:codeNumber | variableType:String<br>variableName:dateOfIssue | |

As understood we can extend the above structure horizontically to include new data objects modelling attributes inserted at run-time, as well vertically to include new classes inserted in the system.

On the other hand instances of classes AdultPerson and IdentityCard store their instance data in the variableValue Dictionary, which their classes inherit from the abstract class Metaclass. The situation is depicted in the following tables:

**Table 2.** Instances of class AdultPerson.

| "height" | obj1 |
|---|---|
| "surname" | obj2 |
| "IdentityCard" | obj3 |

| "height" | obj4 |
|---|---|
| "surname" | obj5 |
| "IdentityCard" | obj6 |

| "height" | obj7 |
|---|---|
| "surname" | obj8 |
| "IdentityCard" | obj9 |

**Table 3.** Instances of class IdentityCard.

| "codeNumber" | obj10 |
|---|---|
| "dateOfIssue" | obj11 |

| "codeNumber" | obj12 |
|---|---|
| "dateOfIssue" | obj13 |

| "codeNumber" | obj14 |
|---|---|
| "dateOfIssue" | obj15 |

In the above picture we can see the mapping between the names of instance variables and the objects representing data for them for each one of the class instances. The variableValue dictionaries can be extended vertically in order to store values for the attributes that will be inserted in the classes. The type of each one of these objects is in accordance with the type of the variable it stands for. We must mention that obj3, obj6 and obj9 are original `IdentityCard` instances. For the rest of the objects that should have the type of Integer or String, this is achieved through the use of the Factory Design Pattern. The hierarchy used is depicted in figure 2 and consists of the following components.

- The Abstract Creator class serving no other purpose than defining the common interface of its subclasses
- The Concrete Creator classes (IntegerCreator, StringCreator, BooleanCreator) with each one of them being responsible for the instantiation of objects having the appropriate type and containing data passed along by the user.
- The CreatorManager class which stores in a dictionary like the one presented in the following table the mapping between different types of variables and the corresponding ConcreteCreator objects.

**Table 4.** CreatorManager Dictionary.

| | |
|---|---|
| "int" | IntegerCreator object |
| "String" | StringCreator object |
| "Boolean" | BooleanCreator object |

- Finally the MetaclassManager class which through its insertValue() method is responsible for the creation of the appropriate data objects.

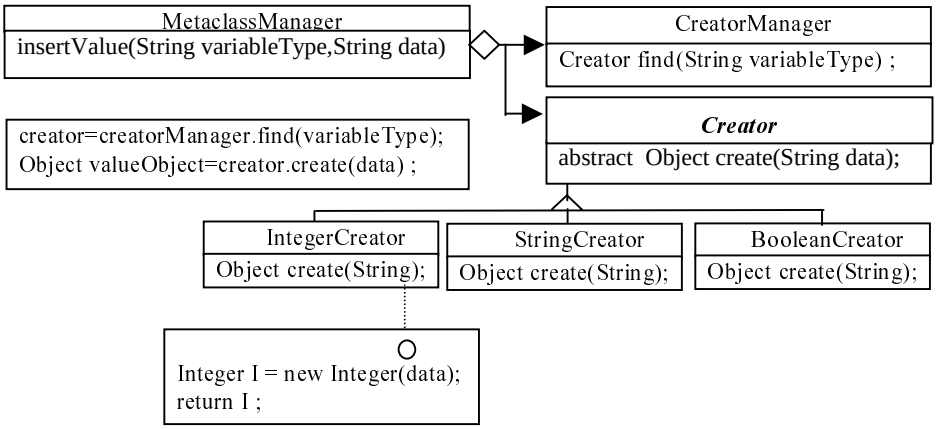



**Fig. 2.** The Abstract Factory Pattern.

The tasks performed in the body of the insertValue() method are the following:

The CreatorManager's find() method is first invoked with a String representing the variable type passed along as argument. The task of this method is to discover the ConcreteCreator instance corresponding to the received argument. That ConcreteCreator instance is then returned to the Metaclass Manager Object.

The create method of the ConcreteCreator instance is subsequently called. An argument representing the value the user wishes to store in the newly created object is passed along to it.

The create method creates an object of the appropriate type and returns it to the MetaclassManager object. After that, the storing of this object within the appropriate class instance takes place.

The insertion of a new attribute in a class using the proposed model could be achieved in the following way.

First a new instance of the Data class should be created. Within its instance variables information about the newly inserted attribute (like its name and type as described before) should be stored. This data object should be stored in the static dictionary containing information about the class properties. The dictionary of our example case after the insertion of the two attributes should have the format presented in the following table.

**Table 5.** variableMap Dictionary after the insertion of variables.

| "AdultPerson" | | | |
|---|---|---|---|
| data1 | data2 | data3 | data4 |
| variableType:int<br>variableName:height | variableType:String<br>variableName:Surname | variableType:<br>IdentityCard<br>variableName:iD | variableType:<br>String<br>variableName:<br>residenc |
| "IdentityCard" | | | |
| data1 | data2 | data3 | |
| variableType:int<br>variableName:<br>codeNumber | variableType:String<br>variableName:dateOfIssue | variableType:String<br>variableName:<br>authority | |

After that all the existing instances of the class should be updated using the before mentioned Abstract Factory mechanism. A new object having the appropriate type and containing the value passed along by the user should be created and stored in the instance dictionary. The format of the AdultPerson and IdentityCard existing instances after the insertion of the new attributes is depicted in the following tables.

**Table 6.** Instances of class AdultPerson after the insertion of variables.

| "height" | obj1 |
|---|---|
| "surname" | obj2 |
| "IdentityCard" | obj3 |
| "residence" | obj16 |

| "height" | obj4 |
|---|---|
| "surname" | obj5 |
| "IdentityCard" | obj6 |
| "residence" | obj17 |

| "height" | obj7 |
|---|---|
| "surname" | obj8 |
| "IdentityCard" | obj9 |
| "residence" | obj18 |

**Table 7.** Instances of class IdentityCard after the insertion of variables.

| "codeNumber" | obj10 |
|---|---|
| "dateOfIssue" | obj11 |
| "authority" | obj19 |

| "codeNumber" | obj12 |
|---|---|
| "dateOfIssue" | obj13 |
| "authority" | obj20 |

| "codeNumber" | obj14 |
|---|---|
| "dateOfIssue" | obj15 |
| "authority" | obj21 |

Until this point we have shown the ability of our approach to support attribute addition. To achieve this in terms of versioning it should be used in combination with a mechanism monitoring the delta changes between the class versions (i.e the attributes to be added or removed from the parent class version when a new one is inserted). This way the programmer would be able to use the version of his choice. Attribute removal in the context of versioning (where no physical removal of the attribute is accepted) could be handled using that monitoring mechanism too. The access to the

values of an attribute being removed from a certain version of a class should be denied when the programmer is using that version.

## 6 Discussion

The proposed technique could potentially support attribute addition and removal in terms of versioning. In this section we briefly present the advantages that it presents in conjunction to inheritance.

- Class explosion and complex hierarchies are avoided.
- No need for object migration.
- Attribute removal becomes an easier task

But there are also certain issues that emerge from its use and should be examined carefully. We let them as part of our future work.

- Substitutability and subtyping provided by inheritance are no longer supported.
- Attribute visibility needs to be considered.
- The behavioural part of a class that also undergoes modifications (because of methods addition and removal) must be treated seriously.
- The monitoring of behavioural changes could provide a means to control evolution in a unified way. This is the case because attribute addition could be simulated at the version control level by pairs of accessor methods for selecting (getter) and updating (setter) attributes. Consequently attribute and method modifications could be handled by the monitoring system in the same way.

## 7 References


1. Casais, E.: Managing Class Evolution in Object Oriented Systems.
2. Cooper, J.: Java Design Patterns - Addison Wesley, Reading, MA, 2000
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Addison Wesley, Reading, MA, 1995.
4. Odberg, E.: MultiPerspectives Object Evolution and Schema Modification Management for Object Oriented Databases - PhD Thesis, 1995.
5. Seiter, L., Palsberg, J., Lierberherr K.: Evolution of Object Behaviour using Context Relations - In Proceedings of the 4th ACM Sigsoft Symposium on the Foundations of Software Engineering, pages 46-57, October 1996.
6. Szyperzski, C.: Component Software - Addison Wesley, Reading, MA, 1997.