

# Object Behavior Adaptation by (Re-)assignment of Property Implementations

Boris Bachmendo  
Institute of Computer Science  
University of Essen  
Schützenbahn 70, 45171 Essen, Germany  
bachmendo@cs.uni-essen.de

**Keywords:** behavior adaptation, object-oriented model.

**Classification:** preliminary work.

## 1 Introduction

The modern software engineering concepts such as separation of concerns and software components require dynamic and flexible adaptation of the behavior of existing objects. In the first case it is necessary in order to integrate modules realizing different perspectives into a single system, in the second one – to adjust the components to the requirements of current environment. By “existing objects” those instances are meant, which are either already loaded into the memory at run-time or whose instantiation is hard-wired in a given (compiled) application code that cannot (or should not) be modified. In both cases the way those objects were created cannot be changed subsequently.

In the class-based object oriented languages like C++ or Java both structure and behavior of objects are unambiguously defined by the class they are instantiated of and only object’s state (but not its behavior) can be changed dynamically at run-time. Several approaches were suggested to overcome this drawback. Besides of the usage of appropriate design patterns [GHJV95] special mechanisms were proposed to influence on the behavior of given objects. The most established of them are:

- *Aspect Oriented Programming (AOP)* [KLM+97] introduces a new kind of objects called *aspects* that can detect predefined interactions (so-called *pointcuts*) between certain methods of objects described by *joinpoints*. Special methods contained in the aspect (*advices*) can be executed before and/or after the originally called method or it can be completely replaced. As aspects are just added to existing program (so-called *weaving*), they modify the behavior of objects externally without changing them.
- *Composition Filters* [AWB+94] intercept and manipulate incoming and outgoing messages according to predefined *conditions*. Since the filters themselves are realized as first-class objects they can be arbitrary attached to given objects and adapt their behavior.

- *Delegation* (also referred to as *object-based inheritance*) [Kni99, VTB98] realizes automatic forwarding of calls to methods not available at the receiver object to its *parent object* with binding of the method's owner variable (usually called *this* or *self*) to the receiver. The parent object is referenced by the special variable of the child and can be dynamically reassigned. Although object's response to delegated messages can be changed this way, the adaptation of the methods owned by the current object still requires its "re-wiring" as explained in [Kni99].

All those approaches try to overcome the inherent limitations of the class-based object-oriented model by extending it with additional constructs. In contrast to them we propose to change the model by allowing the reassignment of method implementations on the object level. Although giving up the traditional class concept we still make use of classes as templates for object behavior and types as definitions of their structure. In this way we propose a compromise between class-based and prototype-based models (whose main drawback is the lack of typing and object classification, cf. e.g. [ZhBi93, Schl96]).

The second sections introduces the main concepts of the proposed model. The third one explains basic operations modifying the behavior of single objects and object groups. The last section summarizes the paper.

## 2 Model Elements

### 2.1 Properties

As mentioned above, in class-based languages object state can be arbitrary changed by assigning values to its attributes. In our model method implementations can be reassigned in the same way allowing the adaptation of objects behavior.

So we generalize attributes and methods, and designate them as *properties* of an object. Each object property has its *implementation* or *body* which is a program block executed as the property is accessed. *Value* is the result delivered by the (implementation of the) property on its invocation.

We consider attributes in the class-based languages as the primitive kind of property implementation whose result is invariant. Such implementation is called *static* in contrast to *dynamic* implementations whose result depends on passed parameters. In our model object's state (i.e. current values of its properties) is unambiguously defined by its behavior (their current implementations).

### 2.2 Types

The set of properties owned by an object depends of the type(s) it belongs to. Each *type* defines a certain interface – a set of related properties. A property is defined by its name, type (i.e. the type of the object that can be returned by property's implementation) and a list of parameters (which can be empty). Properties can be also

declared as *mandatory* (i.e. they have to be assigned by instantiation) or *final* (that are mandatory per se, since their values cannot be changed).

Since a type can inherit from multiple supertypes and each type is a subtype of a special *root type*, the types are ordered by single DAG rather than tree or forest. A subtype has a semantic *is-a*-relationship to each of its supertypes and contains at least an aggregate of their property sets. Supertypes can be also subsequently derived from existing (sub-)types and cannot contain more properties than their intersecting set.

### 2.3 Classes

Classes define implementations of interfaces defined by types, i.e. they contain a set of implementations for their properties. This way the definitions of structure and behavior are separated. Inheritance is also allowed between classes, but it is not a semantic relationship, but rather a way to avoid rewriting of the same code. The three possible application areas of classes are the following:

- *Object instantiation*: Classes can serve as templates for creation of new objects. Different classes can be used for instantiation of single type. A template class cannot have more properties than corresponding type definition and must contain at least all its properties declared as mandatory. If no mandatory properties are defined in a class, an empty object can be created and implementations can be assigned later.
- *Object classification*: After instantiation all object's properties have implementations contained in the template class. So all objects instantiated with help of a certain class belong to the *extension* of this class until any of their properties are reassigned and get the implementation of another class. But classes can be also used to group objects dynamically according to implementations of their properties. Such classes serve as filters allowing to find out objects that contain properties with certain implementations.
- *Object modification*: Classes can be also used to assign a set of related property implementations to a given object. In this case an object is explicitly added to the extension of the certain class. Reassignment of property implementations is described in the next section.

## 3 Reassignment of Property Implementations

### 3.1 Single Properties

In the common programming languages an assignment of an attribute or method of a source object to the attribute of the target object makes the target to reference the result of the invocation of source property. If the source property is also an attribute, the both implementations are identical. This is the only possibility of property reassignment in the class-based languages, because on each invocation the

implementation of the source is automatically evaluated, so that only its value can be accessed. In this section we propose three operations on properties that access and modify their implementations.

The first operation is *copy*. As its result the target property gets the same implementation, that the source property had at the moment of copying. The later modifications of the source do not affect the target property. The target object becomes the new owner of the implementation (i.e. its implicit parameter *this* references the target object), so it is equivalent to delegation. This operation is valid under following conditions:

- The type of the target property must be the same or a *supertype* of the source property type.
- The type of the target object must be the same or a *subtype* of the source object type. This condition ensures that all the possible calls to the properties of *this* can be also accepted by the target object.
- The parameter list of the target property must either contain all the parameters of the source property or the parameters have to be mapped explicitly. I.e. all the parameters of the source have to be listed and either bound the parameters of the target property or to some other values.

Another operation is creation of a *reference* (which is analogue to *compound references* in Smalltalk). The target property always references the *current* implementation of the source. At the invocation of target *this* parameter is still bound to the original source object. The conditions mentioned above are also relevant for this operation. Another distinction between the both operations is, that an implementation can be copied from class to object, whereas a reference to implementation in a class is not allowed.

Sometimes it is necessary to *extend* the current implementation by inserting additional code before or after it. So a new implementation can be assigned that do not completely replaces the old one, but contains a call to the old implementation. The previous implementation can be referenced with help of special implicit parameter (e.g. *old*, analogous to *this* parameter). Subsequent application of this operation results in a concatenation of implementations.

### 3.2 Sets of Properties

In the previous section we described how implementations of single properties can be replaced. Often it is necessary to change the implementation of *multiple properties* of an object correspondingly or to modify behavior of a *set of objects*.

In the first case a class can be assigned to an object, i.e. the object is added to the extension of the class. As a result implementations of all properties contained in this class are assigned to the corresponding object properties. If multiple properties of an object have to be extended in the same way (e.g. in order to add logging of their invocations), the same implementation (that contains calls to their previous implementations) can be assigned to all of them.

We also propose to allow modification of corresponding properties of sets of objects. So behavior of objects of the same type or instantiated with help of the same class can be changed by a single operation.

## 4 Summary

In this paper we described an object-oriented model that tries to combine the advantages of class-based and prototype-based paradigms. Several related approaches were proposed especially for object-oriented databases (cf. e.g. [ZhBi93]). Since the problem of behavior adaptation is particularly acute in the database area due to persistency and long object life cycles.

The main goal of our approach is to allow the dynamical behavior adaptation of single objects and object sets by means of reassignment of property implementations. At the same time we keep established concepts of such class-based languages, such as strong typing and using classes as templates for objects. We strictly separate the definition of object's structure (i.e. types) and its behavior (i.e. classes).

We plan to refine on this approach and implement it as a prototypic object-oriented database management system.

## References

- [AWB+94] Aksit, M.; Wakita, K.; Bosch, J.; Bergmans, L.; Yonezawa, A.: *Abstracting Object Interactions Using Composition Filters*. Proceeding of the ECOOP'93 Workshop on Object-Based Distributed Processing, R. Guerraoui, O. Nierstrasz, M. Riveill (eds.), LNCS 791, Springer-Verlag, 1994.
- [GHJV95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995
- [KLM+97] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.; Irwing, J.: *Aspect-Oriented Programming*. Proceedings of ECOOP '97, LNCS 1241, Springer-Verlag 1997, pages 220-242.
- [Kni99] Kniesel, G.: *Type-Safe Delegation for Run-Time Component Adaptation*. In R. Guerraoui (Ed.): Proceedings of ECOOP99. Springer LNCS 1628.
- [Schl96] Schlegelmilch, J.: *Conflict Resolution using Derived Classes*. In D. Patel, Y. Sun, and S. Patel (editors): Proceedings of the 3rd International Conference on Object-Oriented Information Systems (OOIS'96), London, UK, pages 267-279, Berlin, December 1996.
- [VTB98] Viega, J.; Tutt, B.; Behrends, R.: *Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages*, March 2, 1998. Technical Report at Department of Computer Science, University of Virginia.
- [ZhBi93] Zhao, H.; Biliris, A.: *An Object-Centered Data Model for Engineering Design Databases*. In Proceedings of the Third International Symposium on Databases for Advanced Applications. Daejon, Korea, April 1993, pages 133 - 140.