データベースシステム 103－3
（1995．5．25）

JDMFの核モデルにおけるメソッド

Michael Björn, 穂鷹良介
筑波大学
社会工学研究科
つくば市天王台1－1－1

JDMFのプロセス機能充実のためメソッドを導入する．すべてのモデル概念をオブジェクトと観るJDMFのアプローチを継承して本論文ではメソッドをオブジェクトと観る拡張を行う．メソッドはそのインタフェース（パラメタなど）で特徴づけられるがこれを属性と観れば，一種の属性付きオブジェクトとなる．先行論文で我々はモデルを記述するために無限階層の属性クラスを導入したが，同様の階層を属性とメソッドを統合した特徴（feature）という概念について当てはめることによってモデルを特徴付ける仕組みを提案する．

本提案はオブジェクトの振る舞いそのものの記述をするわけではない．最後に現在までに進んでいるメソッドをオブジェクトと観る実装について述べる．

Methods in the Core Model of JDMF

Michael Björn, Ryosuke Hotaka

University of Tsukuba
Doctoral Program of Socio-Economic Planning
1-1-1 Tennodai, Tsukuba, Ibaraki 305, JAPAN

e-mail: michael@wiz.sk.tsukuba.ac.jp, hotaka@shako.sk.tsukuba.ac.jp
fax: INT+ 81 (0)298 53 5070, **phone**: INT+ 81 (0)298 53 5424

In this paper, we extend the "every modelling concept is an object" approach of JDMF to include modelling of methods. If we identify a method with its interface (or the parameters to the method), we can regard this as a kind of attributed object. In our earlier paper, we described our model using infinite chain of attribute classes. Similar characterization is proposed using the concept of feature that is the integration of both attribute and method concept.

It is however important to note that we still do not model behaviour as such.

We also briefly introduce a prototype implementation which treats methods as objects.

1. Introduction

All systems contain data structures and procedures. In systems written in procedural languages, data structures are defined within procedures making it difficult to manage code for large systems where data dependencies between different procedures usually occur. Object-oriented systems attempt to solve this problem by grouping data structures together with the procedures that operate on them in classes. In the data-oriented modelling approach used for JDMF we take the approach that methods are totally dependent on the data structures they operate on [1]. We concentrate on defining our data structures (classes and attributes) and then write the methods that implement the behaviour of those data structures.

Seen this way, JDMF is a tool for structural definition. For this reason, we have until now primarily only concerned ourselves with class and attribute definitions. Attributes are said to define structure and methods are said to define behaviour. However, all methods have names and parameters which are part of the system's structure (as opposed to the system's behaviour). Thus, if we want to model all structure, we must not overlook the structural part of method definitions, i.e. their names and interfaces (parameters).

We have previously characterized the core model of JDMF as an object-oriented, conceptually "lean" model which is structurally self-descriptive in order to be extensible [2]. As a result of these design decisions, we would like to treat methods as objects, which then in our conceptually "lean" model can be treated in a similar way to other structures in the model, so that we can economize on conceptual constructs.

By concentrating on the similarities between attributes and methods we see that we can model method interfaces as collections of "attributes" in a similar way that we previously have modelled "attributes" of attributes. In this paper we redefine our model with a general Feature class and the classes Attribute and Method as subclasses of Feature.

2. Related Works

There is currently no binary standard for object-oriented languages. The most obvious problem is that different languages employ different and incompatible object models, but a not so obvious problem is that different compilers for a certain language employ different and incompatible linkage conventions. This means that code reuse which has been a buzzword in the object-oriented programming community until now hasn't been realized. Ironically, code reuse is much easier in procedural languages, where formats for Dynamically Linked Libraries (DLLs) have been de facto standardized on various platforms. DLLs provide common linkage formats that enable code sharing across language boundaries.

The Common Object Request Broker Architecture (CORBA) [3] developed by the Object Management Group (OMG) is the first serious attempt to provide inter-language as well as cross-platform compatibility for classes of objects, using a common object model also developed by OMG. One of the first CORBA-compliant implementations is the System Object Model (SOM) 2.0 [4] developed by IBM.

SOM is a technology aimed at producing class libraries, which like procedural DLLs can be shared by many applications. A program can use SOM libraries as fully object-oriented class hierarchies (including instantiation, subclassing, inheritance etc.). SOM decouples the interface and the implementation of a class in the following way:

(1) A standard Interface Definition Language (IDL) is used for structural class definitions in an IDL source file. This includes class name, superclass name(s), attribute definitions and method interfaces.

(2) These structural definitions are then compiled by an IDL compiler for the language specified for the implementation. OMG has so far only specified an IDL binding for C, and SOM also has an IDL binding for C++ but bindings for other languages such as Smalltalk are under development. The IDL compiler generates an implementation template file with a skeleton for the behavioural implementation of the class and two language specific header files with SOM specific bindings for use by the implementation template and program files.

(3) The implementation is then done by filling out the "holes" in the template file with program code using the conventions of the selected implementation language.

(4) The compiler for the selected implementation language is used for compilation of a library which then can be shared by other programming projects.

The implication for the implementation of the next version of the core model of JDMF is that we will use an object-oriented language with an IDL binding (C++) to produce a library version of JDMF. By doing this, we do not need to develop a new programming language for JDMF, but can use any language which has an IDL binding.

3. The Core Model

In this section we update the basic design of the core model and introduce methods as objects.

As in our previous paper [5] we use the symbol "¢" to identify an instance which has a value (such as a class name) which makes it unique in the context under discussion, and the symbols "<<" and ">>" around the name of the class when we talk about an unspecified instance of a certain class

The generalization relationships in JDMF are made explicit in a class called SuperSubRelation. [6] We have updated the class hierarchy of the Core Model in the following way:

0*Object
- Feature*0
  - Attribute
  - Method
- Feature*1
- Feature*2
- Feature*3
- ... (infinitely many)
- SuperSubRelation
- ListObject
- 1*Object
  - 2*Object
    - 3*Object
      - ... (infinitely many)

Since every modelling concept is an object in our model, we describe it by describing the instances of the classes in our model.

For each meta level i (i≥0) we define a base class i*Object, which satisfies the following conditions:

(A) i*Object¢ ∈ : (i+1)*Object¢

(B) (i+1)*Object¢ ⊂ i*Object¢

The instances of the i*Object base classes can schematically be viewed as follows:

Class	Instances	Meta level
0*Object	(No direct instances)	0
1*Object	0*Object¢, SuperSubRelation¢, Feature*0¢, Feature*1¢ ... Feature*j¢, Attribute, Method	1
2*Object	1*Object¢	2
3*Object	2*Object¢	3
	⋮	⋮
i*Object	(i-1)*Object¢	i

*Fig. 1: Overview of instances in the i*Object hierarchy*

The Core Model has infinitely many Feature*i¢ which are orthogonal to the i*Object¢ sequence. Instances of Feature*0¢ characterize the instance features (attributes and methods) of all non-Feature*i classes, whereas instances of any other Feature*i¢ characterize instance features for Feature*(i-1)¢. This means that we can easily extend the characteristics of any Feature*i¢ by making new instances of Feature*(i+1)¢. To separate different classes of feature descriptions, we talk about feature levels.

An overview of how instances of Feature*i¢ characterize features of other classes is given below:

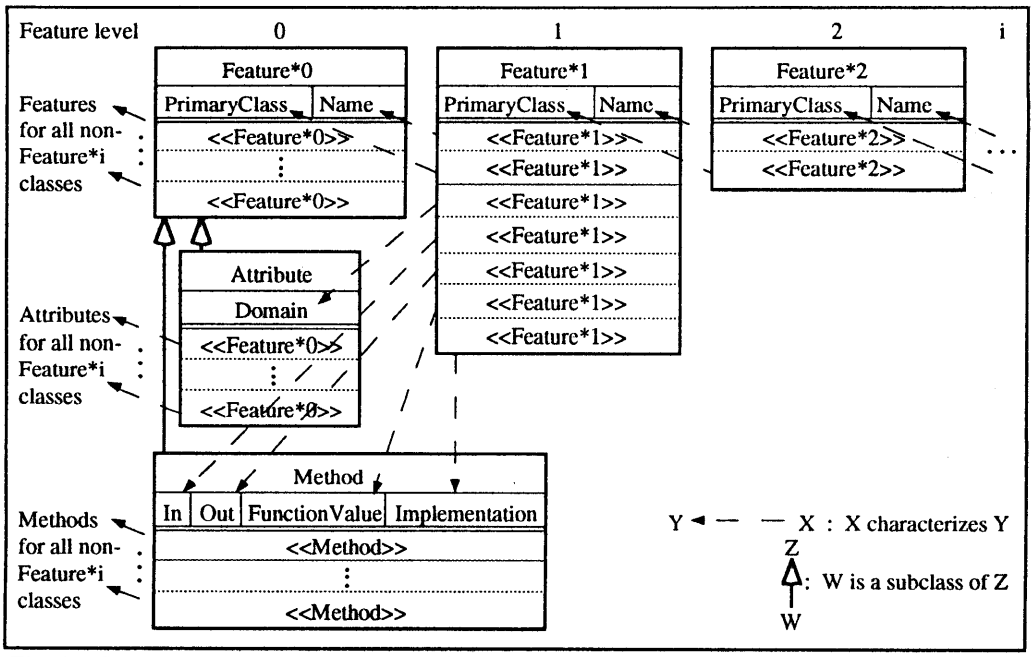

*Fig. 2: Instances of Feature*i classes define features for all classes*

4. The Feature Class

As mentioned in the introduction, all features are instances in the core model of JDMF. The class Feature*0 for this reason provides the basic structure that all features in JDMF will have. It is here important to point out that this is not the only structure that features can have; to the contrary. Since the core model of JDMF will be designed as a class library, users of JDMF will be able to make subclasses of Feature to provide their own specialized features. Such features can have additional structure, and, indeed, additional behaviour.
We have defined two subclasses of Feature, Attribute and Method. We will now look briefly at these two classes.

4.1 The Attribute Class

The Attribute class corresponds to the definition of the class with the same name in JDMF, and to what is normally considered to be an attribute in most other object-oriented models. It will not be further discussed here.

4.2 The Method Class

We will explain the basic method structure as defined in Method, and then go on to discuss the limited number of methods to be provided in the Core Model.

Method has six direct features, which are as follows.

In:	Specifies the list of parameters supplied to the method
Out:	Specifies a list for return values when a method is executed.
FunctionValue:	Specifies a list for return of function values when a method is executed as a function.
Implementation:	Specifies the location of the library which contains the code to be executed.

4.2.1 New and Delete

We have defined two New methods.
The 1*Object¢.New method handles all instantiation at meta level 0 and the 2*Object¢.New method handles instantiation at all other meta levels. The New methods use hard-coded system operations for allocation of memory.

We have defined only one Delete method, in 0*Object¢, which handles deletion of all methods. The Delete method uses hard-coded system operations for deallocation of memory.

JDMF clients can implement their own New and Delete to override the predefined ones. However, all such methods must include a call to the corresponding predefined method to make sure that the basic conventions of the core model are maintained.

4.2.2 Utility methods

To implement the ListObject, we need to provide some methods that allow the user to insert and remove instances. ListObject class corresponds to the definition of the class with the same name in JDMF, and is not explicitly included in the prototype implementation.

4.2.3 Query operations

Initially, we wanted to define queries as methods as well, but since we have found that queries do not implement the behaviour of an object, there is no specific object to which queries belong. For this reason, we will define the query language as a set of operations, and supply it as an external library.We will in another paper define a simple conditional query language based on first-order logic.

5. Calling Methods

We use simple dot notation of the form `object.method(optional parameter list)` for accessing features. The "." (dot) itself represents a hard coded accessor operation.

Let us for example say that we have an Employee class for the employees of a company. The class has an employee_number feature (or more specifically, attribute) which can be accessed through an accessor, for example:

(1) Employee.Show(In: (theEmployee, employee_number), Out: (theEmpNo), FunctionValue: (), Implementation: System/company/
employee.lib)

(2) Employee.Update(In: (theEmployee, employee_number, 37), Out: (), FunctionValue: (), Implementation: System/company/
employee.lib)

The Implementation feature is accessed separately through the CodeLocation method, and can for this reason be omitted from the call. Thus we can rewrite the above as follows:

(1') Employee.Show(In: (theEmployee, employee_number), Out: (theEmpNo), FunctionValue: ())

(2') Employee.Update(In: (theEmployee, employee_number, 37), Out: (), FunctionValue: ())

Furthermore, features which contain no values can be omitted from the call:

(1") Employee.Show(In: (theEmployee, employee_number), Out: (theEmpNo))

(2") Employee.Update(In: (theEmployee, employee_number, 37), Out: ())

When calling a method which has no parameters, we are allowed to drop the optional parameter list totally, if we wish:

(4) theEmployee.retire()

or, equivalently

(4') theEmployee.retire

When combining the dot.notation with the assignment operator "=", we do not need to explicitly use a FunctionValue parameter.

(5) theEmployee.age((FunctionValue: EmpAge))

or, using "="

(5') theEmpAge = theEmployee.EmpAge

In examples (4') and (5') we see that we no longer know if we are dealing with a method or an attribute, which means that we can simply model it as a feature until we know more about how it should be implemented. We do not need to rewrite our interfaces when we later on decide if we want to migrate the feature to an attribute or to a method.

6. Merits of the Feature Class

We have now showed how Features generalize attributes and methods, and how these concepts can be implemented as instances. The reason we do this is that we see three distinct advantages to this approach, namely:

(1) Economy of concepts
By modelling all features as objects, we can keep our model lean and do not need to introduce extra concepts which clutter up the model and make it more difficult to define.

(2) Wider scope of modelling activity
By introducing a Method class we can extend our modelling activity to include method interfaces. It is important to notice that we do not model behaviour as such.

(3) Ambiguity when modelling attributes, derived attributes and methods
When performing object-oriented modelling, practitioners often note that there is some ambiguity. Let us continue with the Employee class example. An employee has an employee_number which he/she is given upon being contracted by the company and which doesn't change until the contract is revoked – employee_number should definitely be modelled as an attribute. Furthermore, an employee might be able to perform_skills according to his/her education and position in the company – it is equally obvious that perform_skills should be modelled as a method.

But suppose we also want to model the employee_age. Usually we think of employee_age as an attribute, but it is in fact a time function derived from employee_birthdate (another obvious attribute) and the present date. Now we suddenly have a number of modelling choices. We can model employee_age as an attribute and manually update the value on the employees birthday; or we can define employee_age as a derived attribute which is automatically updated on the employees birthday; or we could simply model employee_age as a method. Now, this is an extremely simple modelling case, but as the modelling tasks become more complex, the difficulty of differing between attributes and methods increases.

Usually, attributes are said to describe structure of a class (or state of an object) and methods are said to describe behaviour, but we see from the above discussion that this is a truth with some limitations. For this reason, we prefer to think of attributes and methods as describing the features in general of an object. This approach has been taken to its fullest in the language Eiffel [7] where attributes and methods are indeed treated as feauters.

Since Feature is an abstract class with no instances and with semantics that are difficult to explain, we do not explicitly define a Feature class in our model, but the underlying philosophy is the same. Instead we explicitly define infinitely many levels for both attributes and methods, so that the definition of our features can be adapted to our modelling needs.

Let us say that we have an employee object with the value 37 for the attribute employee_age. All we know is that employee_age has a certain state, but not how we it has reached this state. For our modelling purposes it suffices to know that it is a feature.

6. Prototype Implementation

We have made a prototype implementation of the model introduced in the paper. In this section we provide all tables of instances in the model that were output from the prototype.

INSTANCES OF 1*Object (OID: 40389268)				
OID	Class	Delete	ClassName	New
40388264	<<2*Object>>	<<Method>>	0*Object	<<Method>>
40388216	<<2*Object>>	<<Method>>	Feature*0	<<Method>>
40388192	<<2*Object>>	<<Method>>	Feature*1	<<Method>>
40388168	<<2*Object>>	<<Method>>	Feature*2	<<Method>>
40388120	<<2*Object>>	<<Method>>	SuperSubRelation	<<Method>>
40389272	<<2*Object>>	<<Method>>	Method	<<Method>>

*Table 1: INSTANCES OF 1*Object*

INSTANCES OF 2*Object (OID: 40389264)				
OID	Class	Delete	ClassName	New
40389268	<<3*Object>>	<<Method>>	1*Object	<<Method>>

*Table 2: INSTANCES OF 2*Object*

INSTANCES OF 3*Object (OID: NONE)				
OID	Class	Delete	ClassName	New
40389264	NONE	<<Method>>	2*Object	<<Method>>

*Table 3: INSTANCES OF 3*Object*

INSTANCES OF Feature*0 (OID: 40388216)				
OID	Class	Delete	PrimaryClass	Name
40388048	<<1*Object>>	<<Method>>	<<2*Object>>	ClassName
40388036	<<1*Object>>	<<Method>>	<<1*Object>>	Super
40388024	<<1*Object>>	<<Method>>	<<1*Object>>	Sub
40388012	<<1*Object>>	<<Method>>	<<1*Object>>	PrimaryClass
40388000	<<1*Object>>	<<Method>>	<<1*Object>>	MethodName
40387988	<<1*Object>>	<<Method>>	<<1*Object>>	Code
40387976	<<1*Object>>	<<Method>>	<<1*Object>>	Class

*Table 4: INSTANCES OF Feature*0*

INSTANCES OF Feature*1 (OID: 40388192)				
OID	Class	Delete	PrimaryClass	Name
40387944	<<1*Object>>	<<Method>>	<<1*Object>>	PrimaryClass
40387928	<<1*Object>>	<<Method>>	<<1*Object>>	Name
40387916	<<1*Object>>	<<Method>>	<<1*Object>>	In
40387904	<<1*Object>>	<<Method>>	<<1*Object>>	Out
40387892	<<1*Object>>	<<Method>>	<<1*Object>>	FunctionValue
40387880	<<1*Object>>	<<Method>>	<<1*Object>>	Exceptions
40387868	<<1*Object>>	<<Method>>	<<1*Object>>	Implementation

*Table 5: INSTANCES OF Feature*1*

INSTANCES OF Feature*2 (OID: NONE)				
OID	Class	Delete	PrimaryClass	Name
40387836	<<1*Object>>	<<Method>>	<<1*Object>>	PrimaryClass
40387820	<<1*Object>>	<<Method>>	<<1*Object>>	Name

*Table 6: INSTANCES OF Feature*2*

INSTANCES OF Method (OID: 40389272)				
OID	Class	Delete	PrimaryClass	Name
40387620	<<1*Object>>	<<Method>>	<<2*Object>>	New
40387584	<<1*Object>>	<<Method>>	<<3*Object>>	New
40387548	<<1*Object>>	<<Method>>	<<1*Object>>	Delete

Table 7A: INSTANCES OF Method

INSTANCES OF Method (OID: 40389272)				
OID	In	Out	FunctionValue	Implementation
40387620	( )	(<<1*Object>>)	( )	1*Object.New
40387584	( )	(<<2*Object>>)	( )	2*Object.New
40387548	( )	(<<0*Object>>)	( )	0*Object.Delete

Table 7B: INSTANCES OF Method (continued)

INSTANCES OF SuperSubRelation (OID: 40388120)				
OID	Class	Delete	Super	Sub
40387792	<<1*Object>>	<<Method>>	<<1*Object>>	<<2*Object>>
40387784	<<1*Object>>	<<Method>>	<<1*Object>>	<<1*Object>>
40387776	<<1*Object>>	<<Method>>	<<1*Object>>	<<1*Object>>
40387768	<<1*Object>>	<<Method>>	<<1*Object>>	<<1*Object>>
40387760	<<1*Object>>	<<Method>>	<<1*Object>>	<<1*Object>>
40387752	<<1*Object>>	<<Method>>	<<1*Object>>	<<1*Object>>
40387744	<<1*Object>>	<<Method>>	<<2*Object>>	<<3*Object>>
40387736	<<1*Object>>	<<Method>>	<<1*Object>>	<<1*Object>>

Table 8: INSTANCES OF SuperSubRelation

Note that Attribute does not exist in the prototype implementation. This is because the "attributes" used so far are extremely simple and can be sufficiently modelled as instances of class Feature. The class Attribute is first introduced when more powerful attributes are needed, such as attributes with domains (used for strong typing). Since the Core Model is self-descriptive, we can define the class attribute using already introduced concepts.

8. Conclusions and Further Research

In this paper, we show how to model methods as objects. We do this by first recognizing that there is a structural part to methods, namely the method interface (or the parameters to the method). Then we treat this interface as being composed of attributes, which we can model as objects. To combine attributes and methods we replace the infinite chain of attribute classes of earlier versions with an infinite chain of feature classes and introduce an Attribute class as well as a Method class as subclasses of Feature*0 .

We see three main benefits to extending the model in this way:

- Economy of concepts
- Wider scope of modelling activity
- Less ambiguity prbolems when modelling attributes, derived attributes and methods

We also briefly introduce a prototype implementation which treats methods as objects.

In this paper we have assumed that all methods execute without fault. This is however not always the case. When the execution fails, an exception is raised and we need to include some sort of exception handling. Our first approach was to include an exception feature in the Method class, but this was not totally satisfactory. We probably need do introduce exceptions as events, but this needs further study.

Another area which needs further investigation is the definition of accessors. In this paper we have used hard-coded accessor operations which are invoked using the dot notation. It would however be desirable to explicitly include accessors as Get and Set methods so that the user could include constraint checking directly into the accessor methods. When we tried to define accessor methods for the class 1*Object, an infinite chain of accessor calls occurred, so we had to leave out this functionality until we find a better approach.

9. References

[1] H.K. Kim, M. Bjorn, H. Yao, R. Hotaka, "A Sentential Function Mapping Method for Object-Oriented Analysis and Design", Proceedings of First Asia-Pasific Software Engineering Conference, IEEE Computer Society Press, 1994

[2] Hotaka, R. and M. Björn, "Data-Oriented Approach to Business Information Modelling", Proceedings of ICODP '94, North-Holland, 1994

[3] OMG, "The Common Object Request Broker Architecture and Specification (CORBA)", Object Management Group, 1992

[4] IBM, "Object-Oriented Programming Using SOM and DSOM", IBM, 1994

[5] M. Bjorn, H.K. Kim, R. Hotaka, "A Self-Descriptive Conceptual Schema Modelling Facility, its Implementation and Extension, Proceddings of ISCO3, IFIP, Chapman & Hall, 1995

[6] JSA, "A Data Modeling Facility: JDMF/MODEL-1992", Japanese Standards Association: 1-22, 1993

[7] B. Meyer, "Object-Oriented Software Construction", Prentice-Hall, 1988