IEEE Region 10 Conference, Tencon 92
11th - 13th November, 1992
Melbourne, Australia

Design and Implementation of a Programming Language Based on Objects and Fields

Fumihiko NISHIO, Toyohide WATANABE and Noboru SUGIE

Department of Information Engineering,
Faculty of Engineering, Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464–01, JAPAN

Abstract

The object-oriented model is very successful to represent various phenomena in our real world from an entity-category point of view. Many current topics have reported that this paradigm is very appropriate to design and implement information systems. However, it is not sufficient to model the dynamic actions of autonomous entities under this paradigm. This model is in short of representing the relationships between entities and the environments. In this paper, we propose an experimental object-oriented programming language to manage the relationships together, and also report the practical implementation. Our main idea is to introduce the concept of fields, which control the mutual interaction environments among objects.

1 Introduction

The object-oriented paradigm is effective to represent the characteristic properties of distinct entities to be observed in our real world [1]. Many current reports about modeling techniques inform us that the object-oriented model is very powerful and applicable to design and construct information systems [2]. The basic modeling principle is to look upon various kinds/types of phenomena in our real world as the mutual interactions among entities, which are visible individuals in our real world. Such a modeling method based on this principle can specify well many activities and various phenomena in our environments.

The framework of the object-oriented paradigm is based on the ISA relationship defined in the class hierarchy. Such a framework is too simple in comparison with various phenomena of our environments. Under the circumstances that many different objects exist, it is important to manage the mutual relationships among these objects effectively. The ISA relationship is one method for specifying objects from an object-category point of view. However, the concept of groups, which are composed of interrelated objects, is more important to manipulate various objects dynamically because mutually interrelated objects must share their common objective. Namely, we need appropriate constructors, more or less, with a view to modeling our phenomena from an object-interaction point of view, besides the ISA relationship.

Some research papers point out the importance of object-interaction viewpoint. In [3], abstract objects, which are organized from some existing objects, are proposed. The abstract objects represent the behavior among compositive objects as an virtually organized entity, as a whole. Although this abstraction method enables to represent the activity of a group, it is not appropriate for controlling the communication among individual objects in the group. On the other hand, in [4] the concept of fields is introduced. The field is defined as an entirely passive component, which manages the communicable environment among different objects. Contrary to the approach of [3], this field concept is not sufficient to represent the external activities of groups.

In this paper, we propose an experimental programming language to represent successfully the relationships among various entities in our phenomena on the basis of the object-oriented paradigm. The design concept in our programming language is mainly to introduce the notion of fields, in addition to the concurrency control mechanism for objects. The field is an environment to define the operational range in which individual objects with their common goal can interact mutually, and also it represents a group of interrelated objects.

2 Outline of Modeling

The object-oriented paradigm provides a powerful and effective method for modeling the activities and properties of entities in the real world. However, the framework in the object-oriented paradigm is not always adaptable to specify various phenomena to be observed in our real world. Namely, the following restraints are imposed:

1. The message passing mechanism is basically limited to 1-to-1 though our usual communication

0-7803-0849-2/92 $3.00 © 1992 IEEE

method is classified into 4 types of categories: 1-to-1, 1-to-many, many-to-1 and many-to-many;

2. The relationship among objects is derived from ISA though different kinds/types of entities in our real world establish various relationships, corresponding to their activities in the social structures;

3. The concept of environments is not available in the traditional object-oriented paradigm, though individual entities in our real world do not only interact directly with one another but also are controlled indirectly under their common environments.

These constraints are dependent on the fact that only objects are uniquely distinguished as operational entities and the other existing components are not permitted at all. Therefore, we need the other elements to be able to resolve such constraints, in addition to objects. We introduce the concept of fields. The field is an environment to define the operational range among objects. At least, individual objects can interact effectually with each other in sharing their common goal. This modeling method may be provided on the basis of the concepts in the Entity-Relationship paradigm [5]: the objects correspond to entities and the fields do to relationships.

Introducing the concept of field, our modeling method is based on two distinct entities: a static and passive *field*; and a dynamic and active *object*. However, the following problem arises:

4. We can not always resolve our previous discussion by introducing even passive components, in case of modeling various phenomena in our real world. The distinction between objects and fields is very ambiguous. We may not distinguish objects and fields uniquely because the distinction is too dependent on our modeling views.

This problem is observed in the following example. We consider the relationship between a train and persons. The persons in the train feel that the train is an environment (a field) because all information exchanges or propagations are closed in the train. On the other hand, the persons outside the train understand that the train is an object in itself as their distinguished entity. Namely, the distinction between objects and fields depends on the different views for interacted entities. Therefore, it is important to provide a specification mechanism of modeling various views of the phenomena flexibly. Such a specification mechanism must always describe the nested structure among fields in manipulating objects and fields. We illustrate such a framework based on objects and fields in Fig.1. In Fig.1, we observe that fields "A" and "B" construct a nested structure: "A" is enclosed by "B". In the previous example of a train and persons, "A" corresponds to the train as the inside environment and "B" does

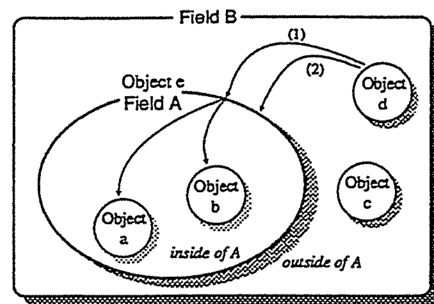

Figure 1: Structure between objects and fields

to the universe of discourse as the outside environment of the train. Also, the train is an object "e" in the outside environment. Objects "a", "b", "c" and "d" are persons in such an example. In "B", "e" as well as "c" and "d" is distinguished clearly as an active entity, and "e" is looked upon as a group of mutually related objects "a" and "b". Here, "a" and "b" do not accept the properties of the object "e", but are controlled passively by the characteristics of "A". This mechanism is very important because the traditional class hierarchy can not support it at all. Therefore, our framework supplies the concept "grouping" in addition to the concept "generalization/specialization" as ISA relationship and "aggregation" as PARTOF relationship [6].

Next, we discuss the functionality of objects and fields, concerning the communication mechanism. The communication among objects is 1-to-1, like the object-oriented paradigm. While, the communication between a field and objects, which are accommodated into the field, is basically 1-to-many from an object to other objects through the field. This communication method is a kind of broadcasting. Namely, the field supports a broadcasting function available for accommodated objects. Here, we must pay an attention to the communication facility of fields for the external objects and internal objects with respect to an field. For example, we consider the objects and fields shown in Fig.1, again. In the field "A" the objects "a" and "b" are internal ones, and the objects "c" and "d" are external ones. It is important that "c" and "d" recognize only "e", as an actual entity, rather than "A". In this case, the message from "c" (or "d") to "e" may be interpreted either as a kind of broadcasting message to "A"(message (1) in Fig.1), or as a direct message to "e"(message (2) in Fig.1).

3 Object and Field

In this section, we address the functional structures of objects and fields.

3.1 Object

The objects in our model are autonomous entities, and can execute their own tasks concurrently. The objects consist of following components.

- *State variables* that represent the current state of object. They are private for the object and are effective to perform methods defined in the object;
- *Methods* that objects perform independently by themselves;
- *A message buffer* that holds messages, sent from other objects. Each object possesses his own message buffer. Messages which were not accepted instantly by the object are temporarily kept in this buffer. This buffer stocks messages into the queue structure according to the arrival order;
- *A message selector* that takes out the first executable message from the message buffer according to the object state.

Users must specify state variables and methods in the definition of objects. The objects are defined in Smalltalk-like syntax.

It is possible to define acceptable conditions for each method. The syntax is as follows:

```
! <ObjClass> guards: #<GuardName> !
<method-name>
    ^ <condition> ! !
```

When `<condition>` is satisfied, the object can execute the corresponding method. Namely, `<condition>` takes a role of the guarded command in CSP [7]. The object is created by a pair of class name and guard name. This enables to create objects which have the same method but different message scheduling policy.

3.2 Field

The field is an environment to make objects active. Previously described, the entity characterized as a field is different from ordinary objects. This differentiation is one of the important aspects in introducing the field. The field has following constructs.

- *State variables* that represent the state for the environment in which objects are exactly available;
- *Procedures* that are called by objects in order to access to state variables in the field. Namely, the objects in a field can change and refer to state variables only through calling these procedures;
- *A message buffer* that holds messages, sent from objects. This message is sent to every object in the field through the broadcasting function;
- *A message selector* that takes out the first queued message from the message buffer;
- *A broadcasting function* that enables to send messages to many unspecified objects within the field;
- *An object list* that records objects which are available in the field. Referring to this list the field can broadcast messages successively. Whenever objects enter into and leave from fields dynamically, this list is updated appropriately.

Users must describe state variables and procedures in the definition of fields. The field is also defined in Smalltalk-like syntax.

Each field has particular procedures such as "`enter`" and "`exit`" in advance. "`enter`" is used for participating the specified object to the field, and "`exit`" removes the object from the field.

3.3 Field as Object

As discussed in the section 2, we must look upon a field as a kind of object when we observe the field from the external view. Therefore, the mechanism for handling a field as an active entity, but not a passive one, is needed. This characteristic is supported applicably by overlaying an object definition onto the existing field definition newly. The definition syntax is as follows:

```
Entity subclass: <ObjClass>
    overlayOn:
        <FieldClass>
    instanceVariableNames:
        <instance variables>
    classVariableNames:
        <class variables>
    poolDictionaries:
        <pool dictionaries>
```

In this definition, the object `<ObjClass>` is overlayed onto the field `<FieldClass>`. The instances of `<ObjClass>` have aspects of object and field.

This mechanism integrates two modeling views: an internal view and an external one. An internal view is defined as a field and an external one is as an object. In the implementation phase, it is able to define these distinct modeling views separately. This separation supports a natural mapping method from our modeling viewpoints to program constructs.

3.4 Communication Methods

Objects perform their own tasks through the mutual communication method. In our communication method, 2 types of communication mechanisms are adaptable: the direct message passing among objects, and the indirect broadcasting in a field. Here, we address the communication method, concerning the synchronization/asynchronization facility among concurrently executing objects.

The direct message passing mechanism among objects is divided into 4 types of communication ways.

1. *Synchronous message transmission*: A sender stops its execution until an appropriate receiver accepts the message. This command syntax is as follows:

   <object-name> <- <message>.

2. *Synchronous message communication*: A sender stops its execution until it receives a reply message from the receiver. This command syntax is as follows:

   <variable> :=
   <object-name> <- <message>.

3. *Asynchronous message transmission*: A sender continues its execution without depending on whether the receiver accepts the message or not. This command syntax is as follows:

   <object-name> <- <message> &.

4. *Asynchronous message communication*: A sender receives a reply message from the receiver asynchronously. This command syntax is as follows:

   <variable> :=
   <object-name> <- <message> &.

On the other hand, the indirect communication mechanism among objects is supported by the broadcasting function of field. The broadcasting function makes it easy to control a group communication in the meeting, the team, etc. The command syntax is as follows:

<field-name> <<- <message>.

4 Implementation

Our experimental language is now implemented in Digitalk's Smalltalk/V running on the Macintosh.

Our programming system is mainly composed of constructors and a parser, on the basis of the class structure in Smalltalk. Constructors are basic programming elements to construct our objects and fields, and are described definitely as the classes in Smalltalk/V. While, the parser is a translator from our language syntax into the Smalltalk description. The classes Entity and Field are constructors to manage the functionalities of object and field classes, respectively. The definition forms for our objects and fields are specified explicitly as the subclasses, using these constructors Entity and Field. Furthermore, the classes MessageHandler and Broadcast are attached to these definitions, as methods for controlling message communication. Actual message communication is controlled by the instances of these classes, but does neither those of Entity nor Field. MessageHandler enables to use various types of communication ways described in the section 3.4. Broadcast supports the broadcasting function of fields. These classes are common to all objects and fields, so we need not redefine.

Fig.2 shows these constructors and the relationships among them. The object is composed of two instances: an instance of Entity (anEntity) and that of MessageHandler (aMessageHandler). The class method create in Entity is invoked when an object is created. These instances which are created simultaneously from Entity and MessageHandler are connected mutually through their instance variables. Also, MessageHandler has an instance variable that manages the message buffer as a property of objects. When aMessageHandler is created, its own process is generated to realize the concurrency of objects. This process picks up an executable message from its message buffer and performs it, if possible. When a message is sent to aMessageHandler, it is queued into the message buffer once. Then, the next message at the top of the message buffer is picked up and then checked its acceptable condition. If the condition is satisfied, aMessageHandler invokes the appropriate method defined in anEntity (and its subclass). If the accepted message is sent synchronously, the reply message is sent to sender. In this way, the message communication is controlled by aMessageHandler. The synchronous/asynchronous control mechanism is transparent for programmers.

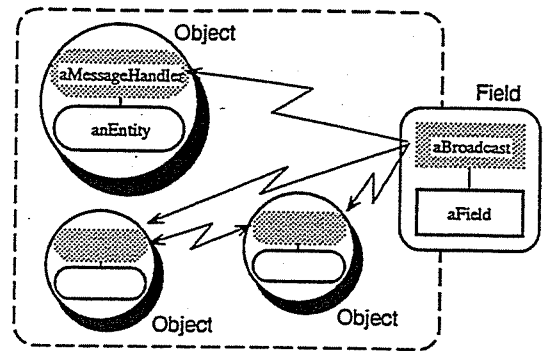

Figure 2: Implementation of objects and fields

Similarly, the field consists of instances of Field (aField) and Broadcast (aBroadcast). aBroadcast maintains the object list that records the member objects of the field. According to this list, messages can be managed by aBroadcast.

On the other hand, the parser is prepared to manipulate synchronous/asynchronous communication ways, as addressed in the section 3.4. This is because the existing facility in Smalltalk can not be always applicable to describe these communication ways directly.

Here, we show a simple example about the finite buffer.

```
Entity subclass #Buffer
  instanceVariableNames: 'buffer n
                          boundary'
  classVariableNames: ''
  poolDictionaries: '' !

! Buffer methods !
```

```
initialize
" initialize a finite buffer. "
    buffer := OrderedCollection new.
    n := 0.
    boundary := 10 !
get
" answer a value in buffer. "
    n := n - 1.
    ^ buffer removeFirst !
put: value
" put a value into buffer. "
    buffer addLast: value.
    ^ n := n +1 !!

! Buffer guards: #BufGuard !
get
" check whether buffer has effective
  data. "
    ^ n > 0 !
put: value
" check whether buffer is full. "
    ^ n < boundary !!
```

The class Buffer is defined as a subclass of Entity, and it has the methods get and put. The guard BufGuard is defined in order to specify an acceptable condition for the methods. For example, the guard get associated to the method get means that the buffer must have one or more value in order to execute the method get.

Instances of Buffer are created and used by the following manners:

```
b := Buffer create: BufGuard.
b <- put: 1234 &.
x := b <- get.
```

In the first example, a finite buffer object is created with BufGuard as its guard. The reference to this object is assigned to a variable b. The second example indicates that a message put is sent to the buffer object b asynchronously. The last example is the manipulating of a synchronous message get. The sender of this message is blocked temporarily until the reply message is sent from b and its value is assigned to x.

5 Conclusion

In this paper, we proposed our experimental programming language based on the concepts of objects and fields. The concepts of objects and fields are very applicable to various problems in the real world. This framework provides a natural way to represent the activities and properties among cooperatively interrelated objects. The group of cooperative objects has two distinct properties. We may not distinguish these properties as object and filed separately, because the distinction is too dependent on our modeling views. So, our framework provides a successful way to use object and field appropriately, corresponding to the modeling views, by overlaying the object onto the field.

In our future work, we must develop an operational interface/environment to support the programming efficiency with a view to making the application ranges clear.

Acknowledgements

We are very grateful to Prof. T. FUKUMURA of Chukyo University, and Prof. Y. INAGAKI and Prof. J. TORIWAKI of Nagoya University for their perspective remarks, and also wish to thank Mr. H. SUZUKI and our research members for their many discussions and cooperations.

References

[1] B.J.Cox: "Object-oriented Programming", Addison Wesley (1986).

[2] B.Meyer: "Object-oriented Software Construction", P.534, Prentice Hall (1988).

[3] R.Helm, I.M.Holland and D.Gangopadhyay: "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", Proc. of ECOOP/OOPSLA '90, pp.169–180 (1990).

[4] T.Maruichi, M.Ichikawa and M.Tokoro: "An Organization Model of Computation and Its Application", trans. on Information Processing Society of Japan, Vol.31, No.12, pp.1768–1779 (1990) [in Japanese].

[5] P.P.Chen: "The Entity-Relationship Model: Toward a Unified View of Data", ACM trans. on Database Systems, Vol.1, No.1 (1976).

[6] A.L.Furtado and E.T.Neuhold: "Formal Techniques for Database Design", P.114, Springer-Verlag (1986).

[7] C.A.R. Hoare: "Communicating Sequential Processes", Comm. ACM, Vol.21, No.8, pp.666–677 (1978).