Caste: A Step Beyond Object Orientation

Hong Zhu and David Lightfoot

Department of Computing, Oxford Brookes University
Oxford, OX33 1HX, United Kingdom
hzhu@brookes.ac.uk dlightfoot@brookes.ac.uk

Abstract. The concepts of object and class have limitations in many situations, such as where inheritance does not offer natural solutions and design patterns have to be applied at a cost to polymorphism, extendibility and software reusability. Proposals have been advanced to generalize the concepts of object and class, such as in Active Oberon. We adapt the concept of agents to software engineering, propose a new concept called caste – a sort of 'dynamic class' of agents, and examine how it helps solve common problems in object orientation.

1 Introduction

Agent technology has mainly been seen as an aspect of artificial intelligence, though it is increasingly seen as a viable approach to large-scale industrial and commercial applications [1]. In our work we disregard the more evidently anthropomorphic aspects of agent (intention, desire, belief, etc.) and concentrate on those aspects that pertain to software engineering.

The concept of an autonomous agent with its own encapsulated data and procedures can be seen as an extension of the object of OO programming. One of the main features of agents lacking from object-orientation is the encapsulation of process with the state and operations, and the explicit description of environment. The language Active Oberon (Zonnon) [2, 3, 4] includes a means of encapsulating a process as an object body. Another weakness of object orientation is the static nature of class structure. In this paper, we investigate how the concept of class can be naturally generalized by a new concept called *caste*, which is roughly a 'dynamic class' of agents.

2 Limitations of class structure

The concepts of class and inheritance are central to object orientation. The inheritance hierarchy of classes enables polymorphism and extendibility in object-oriented programs, and hence helps improve software reusability. However, single inheritance imposes a tree-like hierarchy on classes that may not match reality well. Multiple inheritance, or the generalized definitions of Active Oberon, reduce this restriction. However, a further restriction is that an object must at all times belong to one and the same class. This can pose difficulties if the relationship between an object and its class needs to be more dynamic. For example, a personnel information management system of a university may preclude a student from being enrolled both as an undergraduate and as a postgraduate, but it is quite common at our university for a staff member to be simultaneously a student. If the lifetime of the system is long enough then it will need to model the change of status, for example, when an undergraduate successfully graduates and wishes to continue as a postgraduate. This cannot be naturally modeled by inheritance relations between classes.

The use of an appropriate 'design pattern' [5, 6] helps overcome these difficulties, but at a cost to polymorphism and extendibility. Such difficult design problems inspire us to search for programming-language facilities that can lead to natural solutions while retaining the advantages of object orientation.

3 Agent and caste

We define agents as active and persistent computational entities that encapsulate data, operations and a behavior protocol and are situated in their designated environments [7]. Here, data represents an agent's state. Operations are the actions that an agent can take. Behavior protocols are rules that determine how the agent changes its states and performs actions in the context of its environment. Each agent has its designated environment, which explicitly specifies a subset of the other agents in the system whose behaviors and states will affect the agent's behavior. By encapsulation, we mean that an agent's state can only be changed by the agent itself, and an agent decides its state changes and actions according to its own behavior protocol. As argued in [7, 8], objects are a degenerate form of agents. The structure of an agent is as follows.

- *Agent name* is the identity of the agent, which can be created as a member of several castes.
- *Environment description* indicates a set of agents whose visible actions and states are visible to the agent. As in the formal specification language SLABS [7], an environment description can be in one of the following forms. (1) *Agent-name*: which means that the agent of the name is in its environment; (2) *All*: *Caste-name,* which means that all the agents in the caste are in its environment; (3) *Variable: Caste,* which means that a specific agent in the caste is in its environment, but the agent may change from time to time. Notice that, environment description differs from the import/export facility in that it describes what kind of agents in the system it will interact with at run-time, which cannot be determined at compilation time. It is one of the most important features of agents that distinguish them from objects, including active objects. The environment of an agent changes when other agents join or quit a caste, if the caste is specified as a part of the agent's environment. An agent also changes its environment by joining a caste or quitting from a caste, or changing the values of its environment variables.
- *Variable declarations* define the state space of the agent. It can be divided into two parts. The *visible part* consists of a set of public variables whose values are visible, but cannot be changed by other computational entities in the environment. The *internal state* consists of a set of private variables and defines the structure of the internal state of the agent, which is not visible by other entities in the environment.

- *Action declarations* are in the form of a set of procedure declarations, which defines a set of operations on the internal state and forms the atomic actions that the agent can take. Each action has a name and may have parameters. An action can be one of two types. When executed, a *visible action* generates an event that is visible by other agents. *Internal actions* can only change the internal state, but generate no externally visible event when executed.
- *Body* is an executable code that forms the agent's behavior protocol. As mentioned above, agents are active computational entities. Their dynamic behaviors can be described by the pseudo-code in Fig. 1.

```
BEGIN
    Initialization;
    Loop Body-code Endloop;
END
```

Fig. 1 Agent's behavior

Usually, the body code of an agent perceives the visible actions and states of the agents in its environment, and decides on what action to take according to the situation in the environment and its internal state. An agent can (1) take a visible or internal action; (2) change its visible or internal state; (3) *join* into a caste or *quit* from a caste. An agent's action is not driven by 'method calls' from the outside. This distinguishes agents from active objects.

Caste is a new concept and a language facility first introduced in SLABS [7]. It is a natural evolution of the concepts of class in OO and data type in procedural languages. Just as a class can be seen as a set of objects with the same pattern of data and methods (procedures), a caste is a set of agents with the same pattern of data, methods, behaviors and environments. The concept of a caste contrasts with the class, however, in that an agent may *join* a caste or *quit* from it at runtime whereas an object is all time an instance of one class. This more general view overcomes the above problem in object orientation. The structure of a caste declaration is given in Fig. 2.

```
CASTE name OF caste-names;
    ENV environment-descriptions;
    VAR variable-declarations;
    ACTION action-declarations;
    INITIAL (parameters): Statement;
BEGIN
    Statement (* body code *)
END name.
```

Fig. 2 Structure of castes

A formal definition of the semantics of castes can be found in [9]. Fig. 3 shows an example of caste declaration, *Persons*. In Fig 4, castes *Undergraduates*, *Postgraduates*, *PhD_Students*, and *Faculties* are declared as subcastes of *Persons*; some details are omitted for the sake of space. An agent of caste *Persons* can join the caste *Undergraduates*. By doing so, the agent obtains some additional state and environment variables defined in the caste Undergraduates, i.e. the *Personal_ Tutor* and *Student_ID*. The agent can then quit from the *Undergraduates* and join the *Postgraduates*. Consequently, the agent loses state variable *Student_ID* and environment variable *Personal_ Tutor*, and obtains additional state and environment variables *MSc_Student_ID* and *Supervisor*. Subsequently, it can quit from the caste and join *PhD_ Students*. However, it can join Faculties without quit from *PhD_Students*.

```
CASTE Persons;
    ENVIRONMENT All: Persons;
    VAR   PUBLIC    Surname, Name: STRING;
          PRIVATE  Birthday:DATE;
    ACTION PUBLIC Speak(Sentence: STRING);
    INITIAL (SN, N: String, BD:DATE):
        BEGIN
            Surname := SN; Name := N;
            Birthday := BD;
            Speak("Hello, World");
        END;
BEGIN
   … JOIN(Undergraduates);  …
END Persons.
```

Fig. 3 An example of caste

4 Conclusion

In this paper, we examined the concept of agent and caste as a language facility to extend object orientation. We are aware of a variety of approaches to the problem we addressed in this paper. We believe that the approach proposed here looks promising to overcome the weakness of static object-class binding in object orientation in a nice and natural way.

We are working towards developing a programming language using agents and castes as a core language facility. The intention is to retain the advantages of object orientation and build on its success while offering more powerful and natural means of expression for solutions to commonly occurring problems in software design. We are investigating the implementation of such a language facility, for example, through active objects.

```
CASTE Undergraduates of Persons;
    ENVIRONMENT Personal_Tutor: Faculty;
    VAR   PUBLIC   Student_ID: Integer;
    INITIAL (PT: Faculty):
        BEGIN  Personal_Tutor := PT; END;
BEGIN
    … QUIT(Undergraduates);
        JOIN(Postgraduates); …
END Undergraduates;
CASTE Postgraduates of Persons;
    ENVIRONMENT Supervisor: Faculties;
    VAR PUBLIC MSc_Student_ID: Integer;
    INITIAL …;
BEGIN
   … QUIT(Postgraduates);
        JOIN(PhD_Students); …
END Postgraduates;
CASTE PhD_Students of Persons;
   …
BEGIN     … JOIN(Faculty); ….
END PhD_Students;
CASTE Faculties of Persons;
    …
END Faculties;
```

Fig. 4 Examples of subcastes

References

[1]Wooldridge, M., Weiss, G., and Ciancarini, P., eds. Agent-Oriented Software Engineering II. LNCS, Vol. 2222, 2002: Springer.

[2]Gutknecht, J., Active Oberon for .net, White Paper, June 5, 2001. Available at URL: http://www.bluebottle.ethz.ch/oberon.net/ActiveOberonNetWhitePaper.pdf

[3]Reali, P., Active Oberon Language Report, March 14, 2002. Available at URL: http://bluebottle.ethz.ch/languagereport/ActiveReport.pdf

[4]Gutknecht, J., Zueff, E., Zonnon Language Experiment, or How to Implement a Non-Conventional Object Model for .NET, Available at URL: http://www.bluebottle.ethz.ch/Zonnon/papers/OOPSLA_Extended_Abstract.pdf

[5]Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns,: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

[6]Grand, M., Patterns in Java, Vol. 1: A Catalog of Reusable Design Patterns Illustrated with UML, Wiley, 1998.

[7]Zhu, H., SLABS: A Formal Specification Language for Agent-Based Systems. International Journal of Software Engineering and Knowledge Engineering, 2001. 11(5): pp. 529~558.

[8] Zhu, H. The role of caste in formal specification of MAS. in Proc. of PRIMA'2001, Taipei, Taiwan, 2001, Springer, LNCS 2132, pp.1~15.

[9] Zhu, H., Representation of Role Models in Castes, Technical Report DoC-TR-03-02, Dept of Computing, Oxford Brookes Univ., UK, 2003. (*Submitted to MATES'03*)