Artifact Representation

Using

Object-Oriented Techniques

By

Jeffrey S. Bramwell

Introduction to Artificial Intelligence
CSCI 8456
30 July, 1997

Abstract – Knowledge representation (and reasoning) is a key component of artificial intelligence applications. Several techniques exist for representing knowledge in a computer-based program. Frames are a very popular method for representing artifacts and ideas within a knowledge base. However, object-oriented programming methods are also very popular for solving many types of programming problems, including knowledge representation. Although the implementation of a frames-based system is quite different than the implementation of an object-oriented system, the underlying concepts of both systems are quite similar. In this paper, I will discuss what is involved in creating an object-oriented knowledge base with similar features found in those of frame-based systems. The main purpose of this paper will be to illustrate the ability to represent the same types of data stored in a frames-based knowledge base using object-oriented techniques along with pros and cons.

Knowledge representation is a very important topic in the field of artificial intelligence research. The ability to represent knowledge for some problem domain and be able to search it programmatically is the topic of many artificial intelligence books and articles. There are many techniques that can be used to represent knowledge in software. These techniques usually involve some implementation of one or more abstract data types. These abstract data types can be created using any of several programming languages. Two of these techniques will be discussed in this paper - specifically, the comparison of object-oriented techniques with frame-based techniques.

Before we get into any in-depth discussion on knowledge representation implementation techniques, I will give an overview of object-oriented concepts and then a brief overview of frame related concepts. Then, the frame-based implementation will be discussed followed by an object-oriented implementation based on the same problem.

Object-oriented programming (OOP) has its roots in software engineering and simulation.[1] Therefore, objects in an object-oriented system are designed to act similarly to real-world objects. Objects are made up of attributes (sometimes referred to as instance variables or properties) and methods (object functions or events). For example, a personnel-tracking system may contain an *employee* object. This object may contain attributes such as *job_specification*, *salary*, *date_of_hire*, etc. This object may also contain methods such as *reviewDate*. The attributes of an object define the state of the object. Object methods provide a means of interacting with the object.

Before one can take full advantage of OOP techniques, there are three basic concepts that must be understood: encapsulation (data hiding), inheritance, and polymorphism.

Encapsulation in object-oriented (OO) terms refers to data hiding. Objects allow various levels of data access to be specified. The details of implementation vary among compilers, but the concepts are the same. Typically, there are three different levels of data hiding that can be applied to both attributes and methods: public, protected, and private. Attributes and methods defined as public can be access by any other object within the application. Those defined as protected can be accessed from within the object they were defined and also by descendant (subclassed) objects. Finally, those defined as private can only be accessed from within the object in which they are defined.

Encapsulation has some very important benefits. By encapsulating data and methods, the implementation becomes very "tight." A properly designed object can only be accessed through published interface methods known as an Application Programming Interface (API). This prevents other objects within the system from inadvertently changing an object's state. This is known as the "black box" technique. Encapsulation is not a technique unique only to OO systems. Some non-OO languages allow certain access restrictions to be placed on variables and functions.

Inheritance is the means by which a specific object can inherit another object's attributes and methods. The best way to explain this concept is with an example. Take the previous example of employee. We may want to further break this down into *salaried* and *hourly* employees:

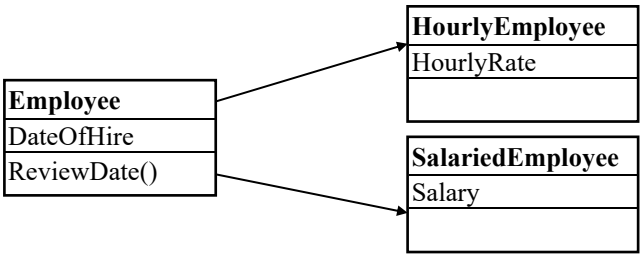

Figure 1

In figure 1 above, the objects *HourlyEmployee* and *SalariedEmployee* are inherited from *Employee*. The object HourlyEmployee would not only contain the attribute *HourlyRate*, but also the attributes and methods of its parent object, Employee.

Inheritance is possibly the most important feature of an OO system. It is also quite possibly the most important feature regarding OOP's use in artificial intelligence applications. It is inheritance that allows objects to define an *is_a* relationship with other objects. In the example above, it can be said that HourlyEmployee is a type of Employee. This will be discussed in more detail later in the paper regarding inheritance among artifacts. Inheritance is also a big player in the last concept to be discussed, polymorphism.

Polymorphism is possibly the most difficult concept in OOP to understand. Polymorphism can be loosely translated as meaning many shapes. Polymorphism is what allows us to perform a type of loose binding among objects and their related attributes and methods.

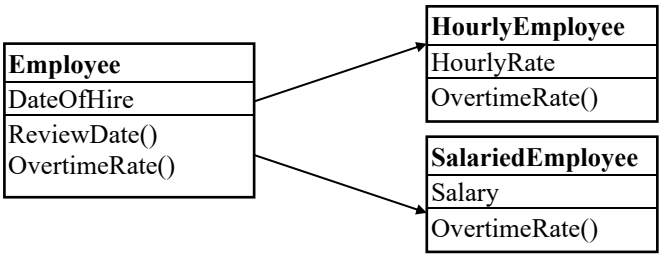

Figure 2

In figure 2 above, both HourlyEmployee and SalariedEmployee inherit the *OvertimeRate* method from Employee. In this case, the Employee implementation of OvertimeRate is what is known as an abstract method. It is not designed to return any meaningful value rather it is designed to be *overridden* by descendent objects. Each descendent would implement its own version of the method. For example, the HourlyEmployee object's implementation might return (HourlyRate * 1.5) whereas the SalariedEmployee object's implementation might return ((Salary / 2000) * 1.5).

So far, we have yet to take advantage of polymorphism. To illustrate this concept, I will provide some example code written in Java.

In listing 1 below, the method *ComputePay* is used to illustrate polymorphism in action. The only parameter defined for this method is of type Employee class. As you will notice in the line marked in bold, the OvertimeRate method is invoked for the parameter *emp*. However, the implementation of OvertimeRate in the Employee class is designed to always return zero. This looks as if we would always end up with a paycheck for zero dollars. Not too good of a deal if it's your paycheck. However, there is more here than meets the eyes. When the call to the OvertimeRate method is executed, emp's class type is determined and the appropriate class method is invoked. For instance, if the code in listing 2, below, was executed, then the implementation for HourlyEmployee's OvertimeRate method will be used. If *temp_emp* had been of type SalariedEmployee, then that object's implementation would have been used instead.

```
public class Employee
{
        Date DateofHire;

        public double OvertimeRate()
        {
                return 0;
        }
}

public class HourlyEmployee extends Employee
{
        double HourlyRate;

        public double OvertimeRate()
        {
                return HourlyRate * 1.5;
        }
}

public class SalariedEmployee extends Employee
{
        double Salary;

        public double OvertimeRate()
        {
                return (Salary / 2000) * 1.5;
        }
}

public class ComputePay(Employee emp)
{
        double pay;
        .
        .
        pay = pay * emp.OvertimeRate();
        .
        .
        return pay;
}
```

Listing 1

```
        double pay;
        HourlyEmployee temp_emp;
        .
        .
```

```
        pay = ComputePay(temp_emp);
        .
        .
```

Listing 2

Now that I have explained the basics of OOP concepts, I will briefly discuss the concepts of frame-based systems and will then compare them to OO concepts.

The basic characteristic of a frame is that it represents knowledge about a narrow subject which has much default knowledge.[3] Frames are made up of slots and slot fillers. Slots are analogous to attributes and sometimes methods in objects. Slot fillers correspond to the values stored in an object's attributes or the values returned by an object's methods. Frames also have the ability to support inheritance as in OOP. However, frames have the ability to be inherited from multiple ancestors if desired. This is a feature not available in most OOP languages.

The Employee class defined above could be defined with a frame as in the following example:

Employee	
DateOfHire	02/24/97
Slot	**Filler**

The above example is a very simple one and not very useful in a frame-based system.

Frames are very useful for defining a hierarchy of relationships through inheritance. For example, the Employee hierarchy could be defined using frames in the following example:

HourlyEmployee	
is_a	Employee
HourlyRate	17.50

SalariedEmployee	
is_a	Employee
Salary	42,500

It is the *is_a* slot that allows us to define inheritance using frames. The is_a slot defines the relationship of one frame with another. For example, SalariedEmployee is a type of Employee. However, this is_a slot could also refer to a set of frames in order to inherit attributes from multiple ancestors.

The Prolog code for creating the above frames might resemble the code in listing 3, below.

```
frame(name(employee),
      isa(null),
      [dateofhire(02/24/97)],
      []).
```

```
frame(name(hourlyemployee),
      isa(employee),
      [hourlyrate(17.50)],
      []).

frame(name(salariedemployee),
      isa(employee),
      [salary(42500)],
      []).
```

Listing 3

In the frame implementation defined in listing 3 above, the first argument denotes the name of the object. The second argument defines the relationship hierarchy. Although the above example only defines one parent in the is_a relationship, it could contain a list of several frames allowing for multiple inheritance. The third argument defines the attributes of the object and the last argument defines any default attributes of the object.

The examples used thus far have been very simple in nature and have not had much relevance with artifact representation, other than the implementation techniques. They were intended to be simple in order to effectively explain the concepts being discussed. The overall goal of this paper is to illustrate the ability to represent knowledge of artifacts and ideas in an object-oriented system. Representation of artifacts and ideas can easily be achieved using frames-based systems. One Prolog implementation is defined in listing 4 below[1]:

```
toolframe(name(artifact, knife),
      isa(artifact, tool),
      [component(knife, blade),
       material(knife, metal)],
      ops(knife, [cutvegatable]),
      []).
```

Listing 4

The above listing describes a knife as an artifact. The knife is a type of tool with two properties: component, a blade, and material, metal. The operations capable by the knife are given as the fourth slot of the frame and no default properties in the fifth slot. This is very similar to the Employee example in listing 3 except that another slot has been added to the frame.

Now that the capability to represent artifacts exists, representing new artifacts based on existing artifacts is relatively straightforward using the property of inheritance. If you wanted to represent a survival knife using frames, your code might resemble the code in listing 5.

```
toolframe(name(artifact, knife),
      isa(artifact, tool),
      [component(knife, blade),
       material(knife, metal)],
      ops(knife, [cutvegatable]),
      []).

toolframe(name(artifact, survival_knife),
      isa(survival_knife, knife),
      [component(survival_knife, [compass, saw]),
```

```
    ops(survival_knife, [cuttree]),
    []).
```

Listing 5

The above listing illustrates inheritance among artifacts using frames. The survival knife frame has all the attributes of a knife (blade component, metal material, etc.) in addition to a few of its own attributes including a compass and saw component and the ability to cut trees.

The question now is how do we represent a knife and its descendant, the survival knife, as an artifact using OOP techniques? Actually, it is not that complicated of a process. The name slot corresponds to the object name, the is_a slot refers to the object hierarchy, the properties slot corresponds to attributes, and the operations slot corresponds to object methods. The default slot can correspond to either attributes or methods. Using this process we can define the knife object described in figure 3.

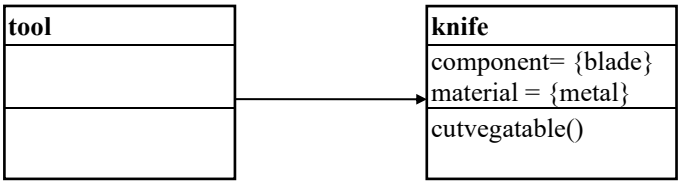

Figure 3

In figure 3 above, the knife object is inherited from the tool object. Knife also has two attributes and one method. So, as you can see, representing an artifact as an object is fairly straightforward. Figure 4 illustrates how the survival knife would be defined using OO techniques. As in the frame example the subclassed object, survival_knife, also inherits its ancestor's attributes and methods. It is this ability to inherit attributes and methods from other objects that allow artifacts to be defined as generically as possible.

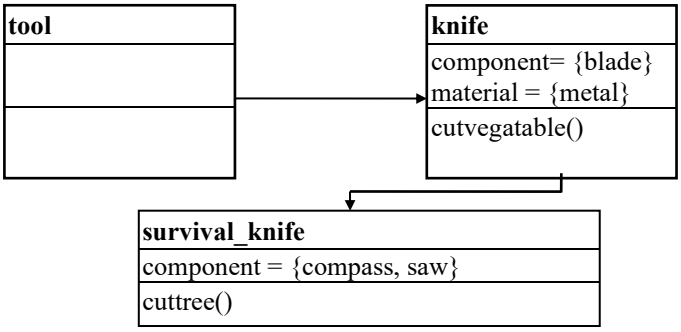

Figure 4

Now that we have the ability to represent artifacts using objects, we need some method for the

objects to interact with one another. Creating rules using Prolog is a very straightforward process because it is built into the system itself. Creating a rules-based system using OO techniques requires a little more work on the programmer's behalf.

Although we have defined relationships within an object hierarchy using inheritance, we need to be able to define relationships among objects outside that hierarchy on an ad-hoc basis. Prolog uses a process known as unification to infer relationships among rules. Unification is not built into OO systems. However, with a little forethought, the relations can be discovered using the OO concepts of inheritance and polymorphism.

By defining a couple standard methods at the top of the object hierarchy, rule-based queries for objects can be created. The is_a relationship already exists between objects in an OO system through inheritance. However, this relationship must be able to be determined at runtime in order to create ad-hoc queries. This can be achieved by implementing an is_a method in the base object and overriding this method for each descendant object. Each object would return its own name for the is_a method. This feature already exists, however, in most OO languages (e.g. typeof). In order to get the name of the ancestor object, you would call the ancestor's is_a method. Note that the name slot of the frame implementation is not necessary in the OO implementation because the class name returned by the is_a method *is* the name of the object. The function performed by the is_a slot in frame implementations is not necessary in OO implementation because you do not have to explicitly traverse the object hierarchy in order to determine the attributes of an ancestor. They are available to any subclassed object at any time (assuming they are not declared with private access).

Also, to create an *affects* relationship between objects, you need to implement an affects method. This allows an object to keep track of other objects it may affect and allows rules to be built based on those conditions. These rules could then automatically fire rules for other objects based upon a specific object's state. For example, if a door switch object is set to the "on" state, then a related door alarm object might be activated. The door alarm object would be referenced by the affects method. This example is illustrated in figure 5. Note that this does not have anything to do with artifact representation using OO techniques. However, it is a further example of how OO techniques can be used when creating rule-based systems.

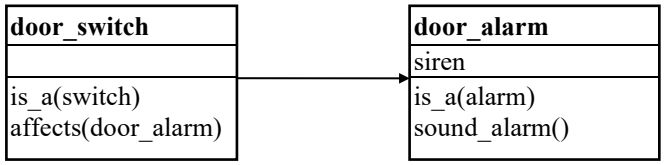

Figure 5

Using the is_a and affects methods, you can build rules similar to:

```
door_switch → door_alarm
```

To put it another way, if the door switch is on then the alarm is on. The inference engine would

traverse a knowledge base based upon the is_a method and fire other object's methods based upon the state of the current object and the affects method.

OOP techniques have several advantages over frame-based techniques. One is the ability to inherit methods and attributes from ancestor classes. Artifact representation lends itself very naturally to this technique. Another advantage is the ability to program in a procedural (algorithmic) language as opposed to a logic-based language. These languages are generally more prevalent than logic-based languages and are widely taught in universities. Also, other OOP features such as encapsulation provide a method for "hiding" the implementation of rules. This is not so easily done in a frame-based system.

On the other hand, frame-based techniques have some advantages over OOP techniques. For example, new frames can easily be created at run-time and asserted into the knowledge base at will. Creating new objects in an OOP language is not easily accomplished if it can be accomplished at all.

It should now be relatively clear that OOP techniques are a valid alternative to the frame-based solution to artifact representation. The purpose of this paper was to give you an understanding of the similarities of knowledge representation, specifically artifacts, between frame-based systems and object-oriented systems. OO concepts have been discussed in some detail and have been compared with similar features in frame-based systems. One major topic, state space search, was not discussed as it is outside the scope of this paper. However, it should be mentioned that any of the abstract data types, such as trees, which can be created in a logic-based programming language such as Prolog, could also be created in OOP languages. Also, any standard search algorithm, such as depth-first-search, breadth-first-search, etc., can also be implemented in OOP languages. By using the techniques described above, effective inference engines can be implemented using OO techniques.

References

George F. Luger and William A. Stubblefield, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, Benjamin/Cummings Publishing Company, 1993.

Working Paper: A simple Prolog program for inventing new artifacts.

Joseph Giarratano and Gary Riley, *Expert Systems: Principles and Programming*, PWS Publishing Company, 1994.

Weber, R. J., and Perkins, D. N., How to Invent Artifacts and Ideas. *New Ideas in Psychology*, Vol. 7, No. 1, pp. 49-72, 1989.

Weber, R. J. Stone Age Knife to Swiss Army Knife: An Invention Prototype. Chap. 12 in *Inventive Minds* (R. J. Weber and D. N. Perkins, eds., Oxford University Press, New York), pp. 217-237, 1992.

Heller, P. and Roberts, S., *Java 1.1: Developer's Handbook*, SYBEX, 1997.

Niemeyer, P. and Peck, J., *Exploring Java*, O'Reilly and Associates, 1996.