DIF8901 Object-Oriented Systems

Object Orientation and Artificial Intelligence

Simon Thoresen

The Norwegian University of Science and Technology

1. Abstract

This essay presents two object based techniques whose inter-relations are hotly debated on various forums on the world wide web. The first is the common object orientation approach, and the other is frame based, knowledge based systems. There are so many similarities between the two, that many people view them as the same.

Today we have object extensions to "AI languages" that can be used as AI tools, and frame systems that tend to include more and more features of OOP. This essay looks for a theoretical framework for "hierarchical structures" of which objects and frames are just two different implementations – and with this, tries to merge frames and objects to obtain a unified paradigm.

In the end, we find that there exists a lanugage that implements such a paradigm already, namely Knowledge Engineering Environment. It is a language that is not widely used, but it proves that you can do *any* implementation, using *any* approach – you only need to adapt a perspective in accordance with the language in use.

A large part of this document is dedicated to explaining to the reader the concepts and how-tos of the different techniques, so that the comparison and conclusion at the end become more understandable.

2. Introduction

The artificial intelligence community has been using object based techniques for many years. The most common use is in frame based, knowledge-based systems. Frames are conceptually very similar to classes and share many of their properties. In addition frame based systems allow rule based reasoning to take place, using the values of attributes (slots) of frames as part of the reasoning process. Frame based systems have become widespread because of the power and flexibility of its representation schemas.

This chapter will briefly introduce the two object based techniques, so that the reader can more readily compare them while reading chapter 3.

2.1. Frame based, knowledge-based systems

Knowledge-based systems (KBS) is a subfield of artificial intelligence concerned with creating programs that embody the reasoning expertise of human experts. In simplest terms, the overall intent is a form of intellectual cloning: find persons with a reasoning skill that is important and rare (e.g., an expert medical diagnostician, chess player, chemist), talk to them to determine what specialized knowledge they have and how they reason, then embody that knowledge and reasoning in a program.

Frame-based systems are knowledge representation systems that use frames, a notion originally introduced by Marvin Minsky, as their primary means to represent domain knowledge. A frame is a structure for representing a concept or situation such as “living room” or “being in a living room.” Attached to a frame are several kinds of information, for instance, definitional and descriptive information and how to use the frame. Based on the original proposal, several knowledge representation systems have been built and the theory of frames has evolved. Important descendants of frame-based representation formalisms are description logics that capture the declarative part of frames using a logic-based semantics. Most of these logics are decidable fragments of first order logic and are very closely related to other formalisms such as modal logics and feature logics.

2.2. Object oriented programming

Object-oriented programming emphasizes the concept of an object. An object is an identified unit which has state and behavior. The state is stored in instance variables (sometimes called member variables). The behavior of an object is affected through operations (sometimes called member functions, selectors or generic functions). When an operation of an object is called, the code to be executed is a procedure called a method. In some object-oriented languages objects are created as instances of classes. The class defines the instance variables, and the methods which are to be executed for each of the operations on instances of the class.

The prime features that give object-oriented languages their power are called polymorphism, encapsulation, and inheritance. *Polymorphism* means that a client is written in terms of the operations it needs, rather than the specific objects it will manipulate; any class of object supporting the needed operations may then be manipulated by the client, without any changes to the client. *Encapsulation* means that the client who uses the operations of an object has no knowledge of what kind of object it is, the layout of its instance variables, how much storage it requires, what additional operations it supports, or any other information that isn't provided by some operation on the object. *Inheritance* is a convenient device by which the implementation of an object can be defined differentially: The objects of a class are the same as those of some other class, except with respect to specified differences.

3. Overview

3.1. Frame based, knowledge-based systems

When one encounters a new situation (or makes a substantial change in one's view of the present problem) one selects from memory a structure called a Frame. This is a remembered framework to be adapted to fit reality by changing details as necessary.

A frame is a data-structure for representing a stereotyped situation, like being in a certain kind of living room, or going to a child's birthday party. Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is about what to do if these expectations are not confirmed.

We can think of a frame as a network of nodes and relations. The “top levels” of a frame are fixed, and represent things that are always true about the supposed situation. The lower levels have many terminals – “slots” that must be filled by specific instances or data. Each terminal can specify conditions its assignments must meet. (The assignments themselves are usually smaller “sub-frames.”) Simple conditions are specified by markers that might require a terminal assignment to be a person, an object of sufficient value, or a pointer to a sub-frame of a certain type. More complex conditions can specify relations among the things assigned to several terminals.

Collections of related frames are linked together into frame-systems . The effects of important actions are mirrored by transformations between the frames of a system. These are used to make certain kinds of calculations economical, to represent changes of emphasis and attention, and to account for the effectiveness of “imagery.”

For visual scene analysis, the different frames of a system describe the scene from different viewpoints, and the transformations between one frame and another represent the effects of moving from place to place. For non-visual kinds of frames, the differences between the frames of a system can represent actions, cause-effect relations, or changes in conceptual viewpoint. Different frames of a system share the same terminals; this is the critical point that makes it possible to coordinate information gathered from different viewpoints.

Much of the phenomenological power of the theory hinges on the inclusion of expectations and other kinds of presumptions. A frame's terminals are normally already filled with “default” assignments. Thus, a frame may contain a great many details whose supposition is not specifically warranted by the situation. These have many uses in representing general information, most likely cases, techniques for bypassing “logic,” and ways to make useful generalizations.

The default assignments are attached loosely to their terminals, so that they can be easily displaced by new items that fit better the current situation. They thus can serve also as “variables” or as special cases for “reasoning by example,” or as “textbook cases,” and often make the use of logical quantifiers unnecessary.

The frame-systems are linked, in turn, by an information retrieval network. When a proposed frame cannot be made to fit reality–when we cannot find terminal assignments that suitably match its terminal marker conditions–this network provides a replacement frame. These inter-frame structures make possible other ways to represent knowledge about facts, analogies, and other information useful in understanding.

Once a frame is proposed to represent a situation, a matching process tries to assign values to each frame's terminals, consistent with the markers at each place. The matching process is

partly controlled by information associated with the frame (which includes information about how to deal with surprises) and partly by knowledge about the system's current goals. There are also important uses for the information obtained when a matching process fails.

3.1.1. Frames and objects

The concept of frames was introduced into AI by Marvin Minsky in 1975. They were a attempt to 'tame' the explosive quality of semantic networks. The frame concept lead to the development of objects orientation. Today it is hard to distinguish frames from objects.

Frames are like classes in OOPS. They have 'slots' (fields) which can contain values or methods, or even values with associated methods. For example, a slot could contain a value which, if changed, invoked some methods as well. Slots can also contain constraints on the values they hold. CLOS, the Common LISP Object System implements many of these frame features.

The slots can also contain links to other frames. This is similar to a semantic network.

3.1.2. Differences from a standard semantic network.

The nodes in a frame based network have much more structure (the slots) than the nodes of a plain semantic net.

The meaning of the allowed links is limited. This limitation improves search efficiency.

The main links are the is-a link (or a kind of -- ako line) and the instance-of link. The is-a link semantic is that of subclass (or subset). The is-a link allows reasoning via inheritance. For example, if one knows that a dog is a mammal, one can infer certain properties of dogs via inheritance since 'dog' is a subclass of mammal.

The other fundamental link is sometimes called instance-of or, in sets, member. This defines a relation between a class and the individuals belonging to the class.

There, may or may not, be other allowed link types in a frame or object system. The point is that, unlike general semantic nets, the number and meaning of the links is restricted.

3.1.3. Classes, subclasses and stereotypical reasoning.

In AI, each basic form of knowledge representation, be it logic, productions, frames or anything else is likely to be associated with some form of human cognitive behaviour. Frames and Objects are no exception to this rule. The inheritance relation, at the core of these KR's is related to the human activity of stereotyping.

Stereotyping is a very efficient way of thinking. Since dogs and horses are both subclasses of mammal, one can save a lot of effort by storing knowledge common to mammals in one mammal class, rather than repeating it both in the horse class and the dog class.

Convenient and efficient though stereotypes may be, they also have a down side. One can jump to incorrect conclusions. For example, among the common characteristics we normally associate with birds is their ability to fly. So if we are told a robin is a bird, we deduce that it can fly, even if nobody tells us specifically that it does so.

But if you are told that a turkey is a bird, you would nevertheless make a mistake if you used inheritance from the bird class to deduce that turkeys can fly.

So with this form of KR and reasoning you must always include the exceptions. If there are too many exceptions, the method breaks down.

3.2. *Object oriented programming*

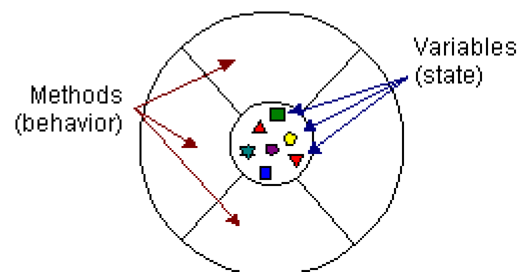
3.2.1. Objects

Objects are key to understanding object-oriented technology. You can look around you now and see many examples of real-world objects: your dog, your desk, your television set, your bicycle.

These real-world objects share two characteristics: They all have state and behavior. For example, dogs have state (name, color, breed, hungry) and behavior (barking, fetching, and wagging tail). Bicycles have state (current gear, current pedal cadence, two wheels, number of gears) and behavior (braking, accelerating, slowing down, changing gears).

Software objects are modeled after real-world objects in that they too have state and behavior. A software object maintains its state in one or more variables. A variable is an item of data named by an identifier. A software object implements its behavior with methods. A method is a function (subroutine) associated with an object.

The following illustration is a common visual representation of a software object:



The object diagrams show that the object's variables make up the center, or nucleus, of the object. Methods surround and hide the object's nucleus from other objects in the program. Packaging an object's variables within the protective custody of its methods is called encapsulation This conceptual picture of an object-a nucleus of variables packaged within a protective membrane of methods-is an ideal representation of an object and is the ideal that designers of object-oriented systems strive for. However, it's not the whole story. Often, for practical reasons, an object may wish to expose some of its variables or hide some of its methods. Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two primary benefits to software developers:

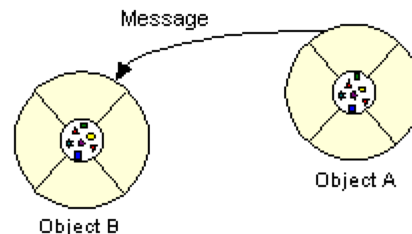
- **Modularity**: The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your bicycle to someone else, and it will still work.
- **Information hiding**: An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it. You don't need to understand the gear mechanism on your bike to use it.

3.2.2. Messages

A single object alone is generally not very useful. Instead, an object usually appears as a component of a larger program or application that contains many other objects. Through the interaction of these objects, programmers achieve higher-order functionality and more complex behavior. Your bicycle hanging from a hook in the garage is just a bunch of titanium

alloy and rubber; by itself, the bicycle is incapable of any activity. The bicycle is useful only when another object (you) interacts with it (pedal).

Software objects interact and communicate with each other by sending messages to each other. When object A wants object B to perform one of B's methods, object A sends a message to object B



Messages provide two important benefits.

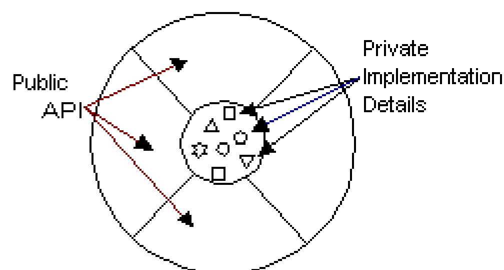
- An object's behavior is expressed through its methods, so (aside from direct variable access) message passing supports all possible interactions between objects.
- Objects don't need to be in the same process or even on the same machine to send and receive messages back and forth to each other.

3.2.3. Classes

In the real world, you often have many objects of the same kind. For example, your bicycle is just one of many bicycles in the world. Using object-oriented terminology, we say that your bicycle object is an instance of the class of objects known as bicycles. Bicycles have some state (current gear, current cadence, two wheels) and behavior (change gears, brake) in common. However, each bicycle's state is independent of and can be different from that of other bicycles.

When building bicycles, manufacturers take advantage of the fact that bicycles share characteristics, building many bicycles from the same blueprint. It would be very inefficient to produce a new blueprint for every individual bicycle manufactured.

In object-oriented software, it's also possible to have many objects of the same kind that share characteristics: rectangles, employee records, video clips, and so on. Like the bicycle manufacturers, you can take advantage of the fact that objects of the same kind are similar and you can create a blueprint for those objects. A software blueprint for objects is called a class.

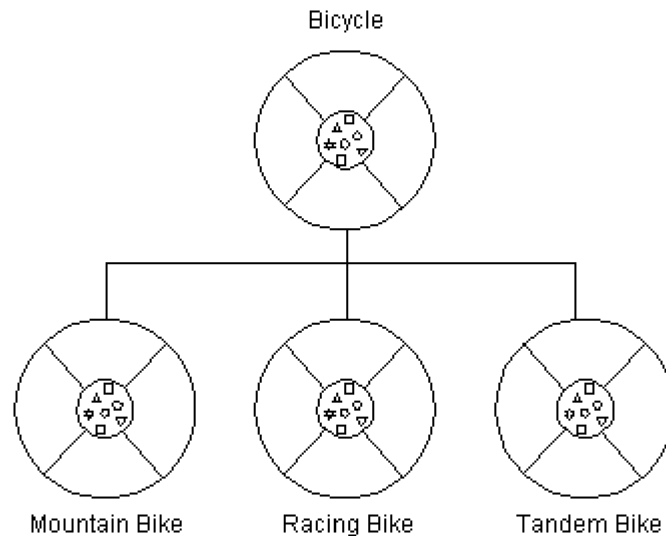


3.2.4. Inheritance

Generally speaking, objects are defined in terms of classes. You know a lot about an object by knowing its class. Even if you don't know what a penny-farthing is, if I told you it was a bicycle, you would know that it had two wheels, handle bars, and pedals.

Object-oriented systems take this a step further and allow classes to be defined in terms of other classes. For example, mountain bikes, racing bikes, and tandems are all kinds of

bicycles. In object-oriented terminology, mountain bikes, racing bikes, and tandems are all subclasses of the bicycle class. Similarly, the bicycle class is the superclass of mountain bikes, racing bikes, and tandems. This relationship is shown in the following figure.



Each subclass inherits state (in the form of variable declarations) from the superclass. Mountain bikes, racing bikes, and tandems share some states: cadence, speed, and the like. Also, each subclass inherits methods from the superclass. Mountain bikes, racing bikes, and tandems share some behaviors: braking and changing pedaling speed, for example.

However, subclasses are not limited to the state and behaviors provided to them by their superclass. Subclasses can add variables and methods to the ones they inherit from the superclass. Tandem bicycles have two seats and two sets of handle bars; some mountain bikes have an extra set of gears with a lower gear ratio.

Subclasses can also override inherited methods and provide specialized implementations for those methods. For example, if you had a mountain bike with an extra set of gears, you would override the “change gears” method so that the rider could use those new gears.

Inheritance offers the following benefits:

- Subclasses provide specialized behaviors from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times.
- Programmers can implement superclasses called abstract classes that define “generic” behaviors. The abstract superclass defines and may partially implement the behavior, but much of the class is undefined and unimplemented. Other programmers fill in the details with specialized subclasses.

4. Programming languages

AI programs have been written in just about every language ever created. The most common seem to be LISP, Prolog, C/C++, and recently Java.

LISP – For many years, AI was done as research in universities and laboratories, thus fast prototyping was favored over fast execution. This is one reason why AI has favored high-level langauges such as LISP. This tradition means that current AI LISP programmers can draw on many resources from the community. Features of the language that are good for AI

programming include: garbage collection, dynamic typing, functions as data, uniform syntax, interactive environment, and extensibility.

PROLOG – It wasn't until the 70s that people began to realize that a set of logical statements plus a general theorem prover could make up a program. Prolog combines the high-level and traditional advantages of LISP with a built-in unifier, which is particularly useful in AI. Prolog seems to be good for problems in which logic is intimately involved, or whose solutions have a succinct logical characterization.

C/C++ – The speed demon of the bunch, C/C++ is mostly used when the program is simple, and excecution speed is the most important. Statistical AI techniques such as neural networks are common examples of this. Backpropagation is only a couple of pages of C/C++ code, and needs every ounce of speed that the programmer can muster.

Java – The newcomer, Java uses several ideas from LISP, most notably garbage collection. Its portability makes it desirable for just about any application, and it has a decent set of built in types. Java is still not as high-level as LISP or Prolog, and not as fast as C, making it best when portability is paramount.9875samhold

The most important aspect, however, is how people use these languages. People with backgrounds in C/Pascal commonly write LISP/Prolog programs using the same algorithm and same control flow. When using these languages like a conventional procedural language, of course the code is likely to be much less readable, less maintenable and less efficient than equivalent C-code. And typically even the compiler gives up any attempt at optimising such programs. As one learns the philosophy of the language and think in that fashion, one feels the difference.

I am sure the same holds for C++ also. I, for one, find it hard to visualise my applications in a totally object oriented form. If I dont follow the object oriented structure and think/program in that form, will my C++ code be any better (efficient, readable, etc) than the same code in any other language?

And in a similar spirit, if I am going to code an algorithm designed for an object oriented paradigm directly, in Prolog/LISP, I would be very likely to be disappointed. I dont think you can move algorithms and programming structure across these different types of languages – but you can probably move ideas. I get the feel that much of the confusion about suitability of languages arises from trying to do this.

5. Comparison

The main differences between frames and objects are:

1. **Syntax**: Frames are usually easier to write than objects, whose syntax depends on the underlying language and is complicated by the greater number of features of object systems. This is, however, a fairly broad classification – the syntax commonly relies more on the language than the system.
2. **Inheritance**: In pure frames the inheritance is dynamic, i.e. values of superframes slots are not propagated; it is the inheritance mechanism which search through the inheritance path each time that an unfilled slot is retrieved. Objects usually complete the instance when it is created, for performance reasons. However, hybrid techniques are today used in frame systems to avoid the time penalty of dynamic inheritance.

   But one should note that in frame systems you normally have hierarchies with respect to specialization, i.e., you have inheritance along "objects" that are more special with respect to their *structure*. In OO systems you often prefer to have hierarchies based on

similar *behaviour*, i.e. if you want to add a new class you add it where you have an object with a similar (slightly more general) behavior but not necessarily a similar structure.

3. **Encapsulation**: In most object oriented languages, data and control are usually encapsulated in 'black boxes' (the “object”), complicating data modification and reasoning. Frame systems don't have encapsulation.

   But, again, this is also just partly true – frame systems are used by access orientend programming, i.e. you access a slot and methods may or may not fire (they are hidden). Objects are accessed by methods and the slots are hidden. You access the structure of frames and you ask for a behavior of an object.

Seeing that all of these differences can be offset by a selective viewpoint, in the end the only difference seems to be origin and purpose – objects are software engineering tools, while frames are AI tools for knowledge representation.

One common problem is that most examples found in AI manuals or research papers are actually better suited to OOP than to frame systems, as the same examples are also used in OOP tutorials – even when frames are used in their context. Except for the matching (recognition) problem, in which an undefined instance is matched against the frame hierarchy to find the class that best fits it description, most of the functionality in frame systems can be achieved with objects. However, some implementations are more natural with frames, other with objects – the choice will only yield different styles of programming.

The OOP approach is, perhaps, more clearly involved with software engineering matters such as encapsulation, reusability, etc. but you can find – or at least simulate – all this with a frame-flavoured language.

Frames and objects are variants of the same principle: object based modelling. Though there are a lot of frames systems, it seems that the following main points can be stated.

Compared to objects, frames offer the following functionalities:

- Access to class level information: essentially slot names and domains.
- Dynamic typing of slots values.
- Event-driven programming, using demons.
- Support for set-valued slots, used to represent relations or cardinality > 1.
- Dynamicity of the creation of classes (usually).

The cost to use these systems is:

- Proprietary models.
- Poor response time.
- Memory consumption.
- No encapsulation.
- Ad-hoc message passing primitives.

Does this mean that frames are a superset of OOP? Indeed, both frames and objects are hierarchical structures with inheritance and slot demons. The textbook answer to this compares only “pure” frames and OO systems. Therefore, frame slots have demons (and no methods) and object have methods (but no slots and no demons).

Does this therefore justify the merging of frames and objects to obtain a unified paradigm?

In Knowledge Engineering Environment (KEE), these concepts *are* merged. It is a frame-based expert system that supports dynamic inheritance, multiple inheritance, and polymorphism. Classes, meta-classes and objects are all treated alike – a class is an instance of a meta-class. The programmer can control rules for merging of each field when multiple inheritance takes place. All methods in KEE are written in LISP. Actions may be triggered when fields are accessed or modified. There is an extensive integration of the GUI and the objects. One can easily make object updates be reflected on display or display selections to update fields. This can in turn trigger other methods or inference rules which may then update other parts of the display.

6. Conclusion

I believe that the difference between frames and objects is mainly historical. Frames go back a long way in AI (at least as far back as Minsky's 1974 paper, and similar things can be found in even earlier work, although without the name “frames”). The first languages with what we might today call “objects” would be Smalltalk and LISP Flavors. These are around the same era as the Minsky paper, but (at least in the case of LISP Flavors) may well have been influenced by Minsky's work.

I hesitate to say that OO is derived from AI, however I do believe that AI at least pushed it along. Today the reason they remain separate is largely terminological. People brought up in the AI schools where “frames” are an important notion will of course stay with that term. For the rest of the world, now that almost every language has a version with OO the term “objects” is more appropriate. Of course, now people in AI are also using those languages (CLOS for example; or even C++) so maybe the term “frame” will disappear in some usage. I think it will stay around, however, because it was meant to capture a psychological model based on focus of thought, stereotypes, and prototypes.

An object-oriented representation is well-suited for representing your world's entities, while a rule-based representation is well-suited for reasoning about these objects. For the sake of this paper, let us formalize the difference as follows:

- Frame is a philosophico/cognitive concept of knowledge representation and its manipulation by *humans*. A frame is a skeletal concept that is to be filled in with details when describing a particular individual later.

whereas

- Object is a logical/programming concept of symbol manipulation by *programs*. An object refers to a program artefact of a specific type (or class) governed by the behaviour prescribed by that type (or class).

Of course there are many fuzzy areas where the two become one, and this leads to a lot of confusion. But when the term “object” replaced “frames” and “actors”, things went really sour!

And finally, the worst part of this mess is that both concepts mean something that is very familiar to everybody, and since everybody has a different perspective on, or flavor of, their meaning you can apply the term to whatever notion you like!