

# A Comparison of Objects with Frames and OODBs

Xindong Wu

Department of Software Development  
Monash University  
900 Dandenong Road  
Melbourne, VIC 3145, Australia

Email: `xindong@insect.sd.monash.edu.au`

## Abstract

Objects and frames are two powerful technologies used in Software Engineering and Artificial Intelligence respectively. Object-oriented databases are currently one of the most important research directions in the database community. This paper discusses and compares the three technologies by looking at their respective characteristics.

## 1 Object Technology

Object technology in software engineering makes it easier to develop, maintain and reuse a wide range of applications. These applications are mainly concerned with data processing. Object orientation attempts to model the behaviour patterns of collections of cooperating physical entities in the real world. Object-oriented programming (OOP) provides a better way of defining data and procedures that are associated with these physical entities than conventional imperative languages such as C, Pascal and Fortran.

OOP was first discussed in the late 1960's when the so called "software crisis" began in large systems development. Methods have evolved since then and have shifted the emphasis from a problem of coding to object-oriented design (OOD). The primary aim of OOD is to improve productivity, increase quality and elevate the maintainability of large software systems [3]. The well defined and widely accepted principles are the concepts of the class, encapsulation, inheritance and polymorphism. At the core of OOD is the class which represents a real world entity by grouping all of its data attributes and procedural operations together into a neatly encapsulated package.

Software productivity is improved primarily by reducing the amount of time required for detecting and removing defects from programming code. Reusing software, in the form of "class libraries" can produce startling increases in productivity and greatly reduce the amount of errors in a large program. However, the emphasis on productivity could have obscured the need for improvements in software quality. Processes that produce high-quality products early in development, such as analysis and design, can greatly reduce

errors discovered later in development such as coding and testing and can dramatically improve software quality [3]. Maintainability, the final objective of OOD, is accomplished by separating the dynamic parts of a system from those parts which are stable. A robust system must be designed with the expectation of change according to the ever changing requirements of clients. Achieving all of these objectives together in a single system is always difficult to accomplish and more than often a trade-off is necessary. With appropriate use, however, the principles of OOD will assist in achieving these goals.

## 1.1 Abstraction and Encapsulation

Abstraction is the principle of capturing useful information by ignoring all the detailed features of an entity that are not relevant to understanding what it does or what it is. Rather than trying to comprehend everything about the entity all at once, we select only part of it.

Abstraction consists of “data abstraction” and “procedural abstraction”. Procedural abstraction can already be found in most imperative programming languages in the form of functions and procedures, which can be used to reduce the complexity of programming code. In OOD, data abstraction is carried out by the definitions of abstract data types (ADTs) – commonly called *classes* or *types* [1]. An ADT is defined in terms of data items and the operations, called *methods* in OOP, that can be applied to these data items. The data within the ADT can only be modified and manipulated by these methods. The resulting notion of encapsulation leads to a separation of interface and implementation. The data of an ADT can only be accessed via the specified interface, while the implementation details such as the operations are hidden and at the discretion of the class implementor or the dynamic decisions of the ADT.

By encapsulation in OOD, each component of a program should hide a single design decision. The interface to each module should be designed so as to reveal as little as possible about its internal implementation details. A language which provides this feature enables the designer to keep related components of a program together in the form of a package in the hope that later changes can be carried out within this package.

## 1.2 Classes

When using an ADT in an imperative language with constructs such as *records* (used in Pascal) or *structs* (used in C) the designer will normally create routines which manipulate the structure. Operations or routines defined for the data structure in one construct cannot be used for another structure in these languages. In an object-oriented programming language (OOPL) such as C++, the data structure and the operations are bound together into one package, called a *class*. A class can have private and public data. The private data cannot be seen or modified by the user without using the public interface, known as *member functions* in C++. This prevents accidental modification of the data and improves code quality by reducing the amount of bugs evident in the code [4].

Variables or instances of a class are called *objects*. There is a fundamental difference between an object and a class: the class is the definition for the data structure, and an object is an instance of that data structure. More than one object can be created from a class definition. This distinction has also led to distinguish *object-based* programming languages from *object-oriented* (OO) ones. In OO languages, objects encapsulate a concrete

data structure and a behavior, and they do not need a type. In OO languages, objects are classified and all objects of the same class share the same behavior. Therefore in an OO language the procedures and functions defined on a data structure are described as part of a class and the class is considered as a generic device for instantiating objects. In this case, the user can use the member functions provided by the public interface of a class to pass messages to objects of the class, and the objects control their own actions and can remember their current state.

Two special member functions, called *constructors* and *destructors* in C++, are provided in many OO languages to allow the user to pass a message to a class and create or destruct an object. A constructor is used to initialise an instance of a class by allocating memory for an array, for instance. Destructors on the other hand are used for clean-up operations, such as freeing any memory the object may have been explicitly allocated.

### 1.3 Inheritance

In an OOP a user-defined class can inherit features of another, thus promoting a much higher level of code *re-use*. Inheritance allows a designer to specify common attributes and services in one class, and then specialise and extend those attributes and services into specific cases. For example, a **manager** class can inherit the properties of a **person** class. One class may also inherit the properties of more than one other class, and this is called *multiple inheritance*.

Single and multiple inheritance is supported by C++ by means of *derived* classes. A derived class is declared by following its name with the names of its *base* classes. A derived class can inherit either the base classes' public parts or both their private and public parts. This is still an issue left open-ended, to be used at the discretion of the designers of individual OOPs. To support multiple inheritance the derived class may form a base class of another derived class permitting the construction of class hierarchies. An inheritance structure is one of the ways of offering reusability, extendibility, lower maintenance cost and of achieving the software engineering goals that designers have been aiming at for 20-30 years [5].

### 1.4 Polymorphism and Dynamic/Late Binding

Although not everyone in the OO community has agreed on it, *polymorphism* is one of the most powerful concepts of OOD. It is the concept of sending a message from one object to other objects in an inheritance hierarchy and invoking the most appropriate behaviour for the object. Consider a **shape** class which can accept the messages **draw** and **erase**. Any derived classes of the **shape** class can receive the same messages. However, with polymorphism, different derived classes such as **circle** or **triangle** would perform different appropriate drawing activities for the corresponding shape without the compiler actually knowing what type of shape it is. Polymorphism presents the property of *operator overloading*. In C++ overloading allows the user to specify member functions with the same name which perform different functions according to how many, and the types of parameters passed. As another example, the addition function '+' may be over-loaded for a variety of user-defined data types (such as matrices, vectors, complex numbers, integers and reals).

To allow the functionality of polymorphism the compiler of an OOP cannot bind the

operation names to programs at compile time [1]. Therefore, operation names must be resolved at run-time. This delayed translation is known as *late* or *dynamic binding*. Using this feature allows the generic ADT `shape` to sometimes be a `triangle` and other times a `circle`. Similarly, overloading allows the same member function to be declared for all of the different derived classes of `shape`. Polymorphism and dynamic-binding are provided in C++ through the use of *virtual member functions*. A virtual function is provided with definition in its base class, but may be redefined in derived classes. That is, a virtual function may have different versions in different derived classes and it is the responsibility of the run-time system to find the appropriate version for each call of the virtual function. Functions that are not marked as virtual may be bound statically to the base class in which it is defined which allows for easier implementation.

There are also other forms of polymorphism than the dynamic binding described above, most of which are available in OOP but also in other languages. *Parametric polymorphism* refers to functions that work in the same way on many different data structures, such as `append` works on lists of small integers and also lists of arrays. Such polymorphism is described by parameterised classes or – in C++ – templates. *Ad hoc* polymorphism is the concept of syntactic or “sugar” overloading, where a programmer introduces some ambiguous notation that is statically resolved by a compiler, for instance by considering the number of parameters. When we wish to distinguish the message passing polymorphism in OO, we speak of *subtype polymorphism*. This terminology suggests that a derived class introduces a subtype of its base class: The instances or objects of the derived class can always be considered as instances of the base class, because all messages for the base class are understood and operated according to the abstraction of the base class.

## 2 Frames

The frame structure was first developed in mid-1970's [7], and has found wide use in Artificial Intelligence and other knowledge based application systems. A frame is a static data structure used to represent well-understood, stereotyped situations. It organises our knowledge of the world based on past experiences. We can revise the details of these past experiences to represent the individual differences for new situations. A frame includes declarative and procedural information in predefined internal relations. The internal relations reflect the semantic knowledge of the specific entity corresponding to the frame. Clearly, any object can be viewed as a specific frame.

A frame can be viewed as a common structure for a kind of entities. It can be used to describe both fixed-pattern entities, e.g., a form feature of a mechanical part, and a conceptual schema of entities, in which some expectations rather than fixed values are included. With those expectations, new data can be interpreted in terms of concepts which are acquired through previous experience. The value of an expectation (often called a slot) can be of any kind of symbol. A symbol may be a list, a string, a char, etc., as well as a data of integer or real. It can also be defined by users as a complex structure which is a combination of symbols, e.g., a pointer to another frame. Two concrete frames from a same schema frame may be quite different in details. Thus the frame structure is more flexible than objects.

A slot in a frame may be a simple description of a macro attribute of the frame, e.g., a value of an integer attribute. It may also be divided into several facets to further express

the property (even dynamic property) of the slot. For example, the “process method” slot in a frame for an electrical or mechanical object may have three facets: (1) default facet which gives a usually used process method, (2) context facet which shows the process method specified during design, and (3) `if_needed` facet which gives knowledge that infers a process method in special cases. The first facet uses the default knowledge about the object. The third facet gives some dynamic knowledge for the value finding of the slot. The dynamic knowledge is often attached in procedure or rule form. When the slot has not been given value during design, the default knowledge and the dynamic knowledge can be used in time.

A frame can be used to describe an object with corresponding default knowledge and dynamic knowledge being attached. Such a frame is called a primitive frame. But a frame with its possible son frames can also be used to describe a part or an assembly in hierarchy structure. Each frame may be a primitive frame or such a higher level frame that is a combination of primitive frames. The hierarchy structure is an intrinsic property of frames.

In summary, frames make it easier to organise knowledge hierarchically. We can describe in a frame an object with its various attributes and other relevant objects and think of the frame as a single entity for some purposes and only consider details of its internal structure for other purposes. Procedural attachment is a particularly important feature. We use procedural attachment to create *demons*, which are procedures that are invoked as a side effect of some other action in the overall system.

### 3 Objects and Frames: A Comparison

Frames are in many ways similar to objects – both have identifiers (or names) and hierarchies, and both have procedures associated with the data slots. Both permit single and multiple data inheritance. However, there are also clear differences between the two technologies.

#### 3.1 Procedure Activation

The procedures of frames, demons, are not directly activated by the programmer, rather they are activated by the situation, i.e., when a data slot is accessed, updated or deleted. Procedural attachments of frames might be defined that automatically perform certain tasks, such as finding an attribute value when none exists, or making sure related attributes are updated when one or the other is changed. This passive structure is in contrast to the methods of OOP that are directly activated by the programmer by message passing. The procedural methods in an object actively respond to messages received from other objects. Also, polymorphism is not offered by frames although one can argue that it could be implemented.

#### 3.2 Composite Frames

A composite frame can contain pointers to other (primitive and/or composite) frames in its slots, and the other frames do not have to be in a specific hierarchy. For example, a chair frame can consist of a back frame and 4 leg frames. This is not allowed in objects. An object can only inherit data and methods from the classes of its higher hierarchy.

Frames and objects both permit single and multiple data inheritance.

### **3.3 Encapsulation and Private Data in Objects**

Frames also differ from objects in their openness. They are designed to work with an inference engine, and their attributes are always open for interaction with any and all pattern-matching rules. This is in contrast to pure objects in which the attributes and methods are so tightly encapsulated, you cannot tell which is which from the outside. Furthermore, the private data in objects cannot be seen by the user.

Objects are a full programming system, designed as much for encoding procedures as data. Frames were never designed to be a full programming system by themselves. Information hiding is a key for objects, and the source of much of the maintainability of object-oriented applications. However, frames have to be open to the inference engine, so whenever any data changes, it knows what rules to activate.

### **3.4 Integration of Objects and Rules**

It seems as if objects, as opposed to frames, and rules are made for each other. Objects are the best way to simulate or model a problem domain. Rules are designed to capture and encode human expertise that is applied to a problem domain. It seems to be very natural to use objects for modeling the domain and rules to represent decision-making applied to the domain.

Further, rules and frames were designed for pure decision-making applications. They are not good for building straight-ahead procedural applications, just as objects and other procedural technologies are not good for building decision-making applications. Integrating rules and objects in a single environment means a developer can easily encode both the decision making and the procedural components of an application.

## **4 Object-Oriented Databases**

Although the history of database systems research is one of exceptional productivity and startling economic impact, many advanced applications have revealed deficiencies of the conventional database management systems (DBMSs) in representing and processing complex objects and knowledge [2]. Object-oriented approaches are currently very popular in processing structurally complex objects while deductive databases or logic databases have been proposed as a solution to those applications where both knowledge and data models are needed.

In object-oriented database systems, complex data structures (e.g. multimedia data) can be defined in terms of objects. Data that might span many tuples in a relational DBMS can be represented and manipulated as a data object. Procedures/operations as well as data types can be stored with a set of structural built-in objects and those procedures can be used as methods to encapsulate object semantics. Containment relationships between objects may be used to define composite or complex objects from atomic objects. An object can be assigned a unique identifier which is equivalent to a primary key in a relation. Relationships between objects can also be represented more efficiently in object-oriented data models by using a more convenient syntax than relational joins. Also, most object-oriented DBMSs have type inheritance and version management as well as most

of the important features of conventional DBMSs. The mandatory features of an object-oriented database, as presented by [1], extend the basic set of OOD principles to include persistence, versioning and integrity control.

Objects in OOP and OODBs are similar in that they require abstraction, inheritance and polymorphism, but there are several important differences. First, database objects must *persist* beyond the lifetime of the program creating them. Second, many database applications require the capability to create and access multiple *versions* of an object. Third, highly active databases, such as those used for air traffic control and power distribution management, require the ability to associate *conditions* and *actions* where the actions are triggered when the constraints are satisfied. Finally, database integrity control demands the capability to associate *constraints* with objects.

## 4.1 Persistence

In O-O database systems memory can be separated into main (*volatile*) memory and secondary (*persistent*) memory. With conventional OOP, objects (particular instances of a class) are stored in volatile memory and whose existence ends with the termination of an application program. In O-O databases, however, objects must be allocated in persistent memory to allow the objects to exist beyond the lifetime of the program which created them. It is immediately obvious that this method of object manipulation will be slow compared to the existence of objects in volatile memory. This problem is regarded as the largest factor of performance degradation in O-O database systems. The implementation of persistence should be transparent to the user where the objects are stored.

In ODE, an O-O database programming language and environment modelled from C++ which offers significant extensions to support database applications [6], persistent objects are referenced by pointers which are local to an application program in which they are declared. Persistent objects are constructed and destructed in a similar manner to conventional objects through the provision of two new operators called *pnew* and *pdelete*. These operators are persistent versions of the C++ heap allocating functions *new* and *delete*. The only difference is that the new functions allocate memory from the secondary device rather than main memory.

## 4.2 Versioning

Many database applications require the capability to create and access multiple versions of an object. Being able to use multiple versions of an object (NOT a class) is important in business situations where the user may need to generate, manipulate and experiment with multiple versions of an object before choosing one which suits their needs.

In ODE versioning is implemented by simply allowing the user to define any number of *version pointers* to a particular persistent object. New objects of that type are created in persistent memory.

## 4.3 Integrity Control

Since many OOPs deal with only volatile stored objects they generally ignore the problem of objects due to faulty programs. However integrity, or protection of data- has always been a major consideration with database technology. Most applications of databases require complex, application specific integrity control, which is beyond the typing capa-

bilities of the language. Thus many O-O database languages including ODE provide the capability to associate explicit constraints with a class definition, as well as triggers which specify actions to be taken in the event that some condition becomes true.

## 4.4 Constraints

Constraints are typically Boolean expressions which are tested after the manipulation of a class. If the operation performed violated the imposed constraint, the operation is *undone* and an exception is raised. A constraint is analogous to an *ASSERT* statement in the SQL language of a relational database system.

## 4.5 Triggers

Triggers are a more generalised form of constraints, where instead of simply raising an exception flag, an action is applied. That is instead of simply rejecting the operation the user may, for instance, be notified that the operation couldn't take place and that they should try the operation again.

ODE allows triggers to be *once-only* in which the action will only be fired once until the user resets the trigger. The other instance of triggers are *perpetual* triggers, where the action will be fired each and every time the condition is satisfied. Again, one can notice the similarity between triggers and the non-standard SQL command *DEFINE TRIGGER* which has been recognised and supported by many conventional relational database management systems.

## 4.6 Query Processing Constructs

Data manipulation in an OODB system is accomplished by means of the operations defined in the class interfaces and through the constructs provided by the programming language surrounding the class definitions. However, ODE provides a high-level query language interface based on SQL. ODE provides mechanisms for iterating over sets or *clusters* of objects. These typically consist of *for* (or *forall*) loops which can be multiply nested. Unfortunately the query processing extension of ODE leaves the efficiency of the query primarily in the user's hands. This contrasts C++'s primary aims: to provide an object-oriented extension to the efficient C language.

## 5 Conclusions

This paper has reviewed the features of object-oriented design in Software Engineering, the frame architecture in Artificial Intelligence, and object-oriented databases. Although frames and objects are similar to each other in terms of hierarchies and procedural attachments, they are different in terms of procedure activation, frame composition, and encapsulation and private data in objects. Object-oriented database systems build database design and management on existing features of object technology, but more features such as object persistence, versioning and integrity control need further attention.

## References

- [1] Atkinson, Bancillon, De Witt, Dittrich, Maier, and Zdonok, The Object-Oriented Database System Manifesto, *Building an Object-Oriented Database System: The*

*Story of O2*, F. Bancilhon, C. Delobel, and P. Kanellakis, Eds., Morgan Kaufmann Publishers, 1992, 3–20.

- [2] R.G.G. Cattell *et al.*, Next Generation Database Systems, *Communications of the ACM* (special section), Vol. 34, No. 10, 1991.
- [3] P. Coad and E. Yourdon, *Object-Oriented Design*, Yourdon Press, 1991.
- [4] B. Eckel, *C++ Inside & Out*, McGraw-Hill, 1993.
- [5] B. Henderson-Sellers, Object-Oriented Information Systems: An Introductory Tutorial, *The Australian Computer Journal*, **24**(1992), 1: 12–24.
- [6] J. Hughes, *Object-Oriented Databases*, Prentice-Hall, 1991.
- [7] M. Minsky, A Framework for Representing Knowledge, *Readings in Knowledge Representation*, R.J. Brachman and H.J. Levesque (Eds.), Morgan Kaufmann, 1985.