# Object-Oriented Data Model

## Table of Contents

1. **Abstract Data Objects**
   - Introduction: 3
   - Abstract Data Objects: 4
   - Methods: 7
   - Messages: 9
   - Summary: 11

2. **Data Classes**
   - Introduction: 12
   - Data Classes: 12
   - Definition of Private Memory: 17
   - Definition of Methods: 18
   - Summary: 20

3. **Dynamic Binding and User Interface**
   - Introduction: 22
   - User Interface and System Data Classes: 22
   - Dynamic Binding: 24
   - Summary: 26

4. **Static Inheritance**
   - Introduction: 28
   - Static Inheritance of Properties: 28
   - Abstract Data Classes: 32
   - Definition of an Object-Oriented DBMS: 34
   - Summary: 34

5. **Dynamic and Multiple Inheritance**
   - Introduction: 36
   - Dynamic Inheritance: 38
   - Multiple Inheritance: 39
   - Summary: 41

6. **Object Identity and Database Query**
   - Introduction: 43
   - Object Identity and Addressability: 44
   - Query Expressions: 46
   - Summary: 49

7. **Metalevel Facilities and Database Architecture**
   - Introduction: 51
   - Metavariibales and Metaclasses: 52
1 Abstract Data Objects

1.1 Introduction

Object-oriented systems are currently receiving much attention and making great impacts in many areas of computer science. They have their roots in programming, as an alternative approach to procedure-driven programming, and is reflected in the development of such programming languages as Simula, Smalltalk and C++. It has since been adopted and extended, particularly to cover the broader range of software engineering activities including modelling, specifications and design phases of software construction. Even the field of artificial intelligence, especially knowledge engineering, have (somewhat independently and in parallel with development in programming) found the object-oriented approach to be particularly effective.

Somewhat a later development is the application of the object-oriented paradigm to databases and database management systems. This interest was perhaps fueled by requirements in new areas of database applications - in particular, hypermedia systems. Such applications call for data modelling capabilities not supported by traditional models of databases or current implementations of database management systems (such as relational or network data models and DBMSs based on them).

![Database Architecture](image)

Figure 1-1 Database Architecture

Conceptually, database systems are based on the idea of separating a database structure from its contents. This was explained in section 1.3 (see also Figure 1.5 there). To briefly recapitulate, the database structure is also called a schema (or meta-structure - because it describes the structure of data objects). A schema describes all possible states of a database, in the sense that no state of the database can contain a data object that is not the result of instantiating an entity schema, and likewise no state can contain an association (link) between two data objects unless such an association was defined in the schema.
Moreover, data manipulation procedures can be separated from the data as well. Thus the architecture of database systems is portrayed as shown in Figure 1-1.

The axioms of conventional data modelling are:

1. Attributes, data objects and relationships belong to predefined types;
2. The schema or metastructure of a database must be specified in advance;
3. Data manipulation facilities are based on a propositional calculus (allowing comparisons of attribute values)

### 1.2 Abstract Data Objects

The principal idea behind object-oriented approaches is that of *encapsulating data in abstract data objects*, or ADO for short (the use of this term is practically synonymous with that of abstract data types, or ADT, which is also commonly used in the literature; where no confusion can arise, however, and when it results in better reading, we will simply use ‘object’ or ‘data object’ to mean an ADO). An ADO has the following properties:

1. It has a unique identity.
2. It has a private memory and a number of operations that can be applied to the current state of that memory.
3. The values held in the private memory are themselves ADOs that are referenced *from within* by means of variable identifiers called *instance variables*. Note the emphasis “*from within*”, which underlines the idea of encapsulation, i.e. such instance variables or objects they denote or any organisation of the objects into any structure in the private memory are not visible from outside the ADO.
4. The only way that the internal state of an ADO can be accessed or modified from outside is through the invocation of operations it provides. An operation can be invoked by *sending a message* to it. The message must of course contain enough information to decide which operation to invoke and provide also any input needed by that operation. The object can respond to the message in a number of ways, but typically by returning some (other) object back to the message sender and/or causing some observable change (e.g. in a graphical user interface).

Operations of an ADO are also referred to as *methods*. Not all methods have to be visible, however - some methods may be needed only internally and, like the structure and contents of private memory, are hidden from outside view. Those methods that are visible externally are called *public methods* and constitute the *public interfaces* of the object. Users or clients of the object need only be aware of its unique identity and its public interfaces to be able to use it.

These properties of an ADO may be pictorially depicted as in the figure below:
Object-Oriented Data Model

For example, the ADO with identity “Person Nick” may be depicted as in Figure 1-3. This object represents a particular person and its private memory will contain values pertaining to that person. These values are accessed and manipulated only through the public interfaces. Thus, the message “Get-Salary” will invoke the corresponding method which will retrieve and return the person’s salary. The “Set-Salary” message on the other hand will invoke the corresponding method to modify that value in private memory representing the person’s salary.

Note that as a user or client of this object, we have no knowledge of, nor do we need to know or care about, its private memory structure. The salary, for instance, may be a value stored explicitly in the object’s memory, or computed dynamically using other values (such as daily rates and number of days worked), or retrieved from some other object. What matters to the client is only the public interface.

Much of the power of the object-oriented approach lies here in data encapsulation. It means that the implementor of some ADO is free to choose any implementation structure he/she deems appropriate, or change it later, say, for greater efficiency. As long as the agreed public interfaces remain the same, clients will be assured of the same (perhaps improved) service. Changes may also add new functionality, ie. new public interfaces. Again, as long as the interfaces used by existing clients are maintained, they would not be affected. The extended object, however, may take on new clients that exploit the new interfaces.
As implementors of an ADO, however, we must know how private memory is structured and organised. In principle, and in keeping with the object-oriented view of values, private memory is simply a collection of other ADOs. More specifically, it is a collection of named memory locations. These names are *local* to (ie. unique only within) the ADO in question. At each of these named locations, we may store the identity of some other data object. These, in contrast, are unique and global to the database. For this reason, the local names are referred to as *instance variable names* or simply *variable names* (‘variable’ because the location’s contents may change, and ‘instance’, as we shall see later, is synonymous with ADO). Arbitrarily complex associations between objects may therefore be constructed through their memories.

Consider, for example, the collection of objects in Figure 1-4.

![Figure 1-4 Object collection with their private memories](image)

The schematic on the left depicts the situation we wish to represent in object-oriented terms, viz. there is a department of computer science with a collection of employees (two are shown). Each employee has a number of attributes (the ‘Name’ attribute is shown).

The schematic on the right depicts one possible representation, which comprises three data objects. Each object has a unique identity (‘DCS’, ‘Alex’ and ‘Nick’ respectively) and a private memory which contains a collection of instance variable names and their values (eg. in the ADO ‘Alex’, the variable ‘Affiliation’ has value ‘DCS’). The public methods of these objects are unimportant for now and are omitted.

Note that the value of a variable is in effect a reference to an ADO, using the object identity rather than a copy of the object. This “reference semantics” of object containment means that a particular object can be referenced from within many other objects, ie. a form of re-use of data objects. Thus, each of the objects ‘Nick’ and ‘Alex’ refers to the object ‘DCS’ as its affiliation. The object ‘DCS’ in turn has both references to ‘Nick’ and ‘Alex’ in its private memory. Together, these associations capture the relationship (expressed in the left schematic) between a department and its employees.

Of course, variable names may be arbitrarily chosen. The names, in themselves, do not constrain the values they may contain and the same data object may be re-used in
different variable names of different objects. So another ADO, say ‘University’, may have a variable named ‘Departments’ whose contents are a collection of references to department objects. The data object ‘DCS’ can then also be a value in this collection.

### 1.3 Methods

An ADO’s methods are code that operate on its private memory in response to an incoming message. As we have seen above, private memory is a collection of other objects. Thus, a method basically achieves what it needs to do by sending messages in turn to appropriate objects in the private memory. This is illustrated below.

![Method Behaviour Diagram](image)

1. Method 2 is activated by an incoming message
2. It in turn invokes appropriate objects in private memory by sending each a message it puts together (possibly using values in the incoming message)
3. Invoked objects eventually return responses
4. Method 2 collects responses and compose a response that is directed back to the sender

**Figure 1-5** Method Behaviour

For example, suppose the ‘DCS’ object responds to a message ‘GET_NAME’, responding with a text string denoting the name of the department. This is shown in Figure 1-6.

![Example Object Diagram](image)

**Figure 1-6** The example object ‘DCS’ responding to a message

Suppose further that the object ‘Nick’ has a method called ‘WORKS_FOR’, intended to return the name of the department that Nick works for. Of course, this information is contained in the object ‘DCS’ in Nick’s private memory. So the method ‘WORKS_FOR’ may be implemented by simply sending the message ‘GET_NAME’ to ‘DCS’, waiting for
the response and then relaying this back to the sender. This is illustrated in the following figure.

![Figure 1-7 Relegating (part of) the work to other objects](image)

If methods achieve their work by sending messages to other objects, and these objects in turn send more messages to yet other objects, and so on, when will any result be generated in response? The answer is that there are *system-defined objects* whose internal structure or method definitions are not our concern. What is important about these system objects is that they provide a number of public interfaces that guarantee a response if invoked with appropriate messages. The simplest type of system objects behave like variables in conventional programming languages, i.e., they have only one variable in their memory and provide public interfaces such as ‘GET_VALUE’ and ‘SET_VALUE’ that respectively reads and sets the variable. The ‘Name’ variable in Figure 1-7 could presumably hold such objects.

In applying the object-oriented paradigm to databases, the ADOs are the principal units of data populating the database. ADOs may be created and once created will persist until they are explicitly deleted. They exist independently of particular user sessions, and different sessions may access or modify them.

The following illustration shows a database of three objects on the left. Assuming that the object ‘DCS’ was sent a ‘DELETE’ message, that object will cease to exist. The outcome is to remove ‘DCS’ from the database. Note that in this case, the consistency of the database is also maintained by removing any use of ‘DCS’ in the private memories of other data objects.
1.4 Messages

We have talked about messages above rather loosely. Public methods of objects must clearly be formal, however, and will only recognise messages that are appropriately structured and carrying the right sorts of data. Sending a message to an object is not unlike calling a function or procedure in conventional programming languages. So we may expect that the message must specify

1. the method name that should respond to the message, also called the ‘selector’ component of the message,
2. the object to which the message is directed, also called the ‘target’ or ‘receiver’ of the message, and
3. the actual parameters (of the right sorts) for the method’s code to operate on. Parameters are themselves ADOs.

This message structure is illustrated in the figure below.
Actual parameters in a message are optional, i.e., some methods do not need input parameters and compute their responses only from the internal state of the object. In these cases the message structure comprise only a selector and a target.

A method may send back some value in response to a message, or it may not. This will depend on the problem domain and how we choose to design our methods. In the case of the “Set Salary” method above, no return value is necessary - only the effect of setting the ‘Salary’ value is important. A method that does respond with a value actually returns a data object. This is illustrated in the Figure 1-10. The message “Get Salary(Nick)” retrieves the object in the ‘Salary’ variable and passes it back in a return message to the sender.

It is important to notice that since the response to a message (when there is one) is itself a data object, it can be the target of another message. In such cases, we may treat messages in much the same way as we do functional expressions, i.e., a message may be viewed as a function denoting a data object and can therefore be used where a data object is a valid expression. For example, assuming that “Print” is a public method of the data object “2000” in the above example, then the following is a valid message:

```
Print( Get-Salary(Nick) )
```

That is, the message “Get-Salary(Nick)” evaluates to the value returned, which is the data object “2000”, which then becomes the receiver of the message with selector “Print”.
1.5 Summary

We have introduced:

1. ADOs as principal components of an object-oriented database. Each ADO has an identity, its own private memory and responds to a number of messages. A message is always directed to a particular ADO and is in effect a request to carry out one of its operations.

2. The public interface of an ADO is the set of operations or methods that may be invoked through sending appropriately structured messages to it.

3. A method is some code that prescribes the processing to be undertaken when it is invoked by an incoming message. This processing will typically involve sending further messages to data objects in private memory.

4. A system object is a pre-defined ADO. Its internal memory structure or methods definitions are hidden and unimportant to us. They are otherwise like any user-defined ADO and may be used by application-specific objects.

Unlike relational databases, the data structures of objects in an object-oriented database are encapsulated (hidden) and cannot be manipulated directly by generalised procedures. This is because generalised procedures are only possible if the data structure is known and uniform over all objects, e.g. the relation or table in the relational model is a known and uniform structure, allowing generalised procedures such as query operations to manipulate tabular structures independently of actual contents. Instead, each ADO presents a public set of methods that operate over its private data structures. This allows great flexibility in the design and definition of objects while at the same time allows object capabilities to be shared and re-used amongst persistent objects.

ADOs are values, in much the same sense that ‘5’ is an integer value or that a particular set is a relational value. They constitute the contents of an object-oriented database. A separation of structure and content may be achieved by describing classes of ADOs, in much the same way that a relational schema describes a set of tuples. A particular ADO will then be an instance of some class description, which will define the memory structure and methods common to all ADOs in the class. This will be the subject of the next chapter.
2 Data Classes

2.1 Introduction

From the discussions of the previous chapter, the basic architecture of an object-oriented database may be illustrated as in Figure 2-1 below. The persistent data of tables in the relational model is replaced by a collection of persistent ADOs, and the generalised data manipulation procedures are replaced by the public interfaces of objects.

![Figure 2-1 Object-Oriented Database Architecture](image)

The ADO concept of encapsulating data and operations over them, however, poses a problem in the database creation. The creation of an ADO, by its nature, requires private memory to be defined and methods associated with it. But typical databases will have thousands of objects, and having to define each object individually is clearly impractical.

2.2 Data Classes

To avoid such problems, object-oriented systems introduce the concept of Data Classes. A data class is a description of a number of similar ADOs. In other words, each ADO belongs to a particular data class, or equivalently, is said to be an instance of that data class. The ADOs of a class are ‘similar’ in the following sense: they have in common the same public interfaces and private memory structure (see Figure 2-2). They are not the ‘same’, however, because each may have different memory contents and thus behave differently in response to the same message. A data class description is therefore a metastructure description, separating structure from content, in much the same way that a relational schema was a metastructure in the relational data model.
Content is introduced by creating instances of data classes. For this purpose, every data class has an implicit method called ‘NEW’. In response to a message with selector ‘NEW’, a data class will return a unique object that has the private memory structure and methods as described by the data class (excluding, of course, the method ‘NEW’ itself). Subsequently, messages may be sent to such objects invoking any of their public methods to manipulate internal data (eg. insert, remove, change, etc). A data class may thus be viewed as a template, or a ‘cookie cutter’, or even as a ‘factory’ for producing ADOs (ie. the instances).

Figure 2-3 shows this role of a data class. In response to each ‘NEW’ message a unique instance is created. Also illustrated is the sending of the ‘Print’ message to the instance ‘Nick’ and to the instance ‘Alex’. Both instances can respond to this message, since each was created in the image of the same data class description which has the corresponding
Object-Oriented Data Model

method definition. The results they return, however, may be different since each instance’s private memory is independent of one another.

Besides user-defined data classes, an object-oriented system typically includes a number of system classes. In fact, user-defined classes must in the end build upon system classes. We may assume that a system class, like any data class, defines a private memory structure and public methods. The private memory structure is of course hidden and inaccessible to user-defined classes and objects. What is important, though, is that instances of system classes, or system objects, may be used (through their public interfaces) by other objects.

As an example, a typical system class might be the class called ‘Numeric’. Instances of this class are intended to denote particular numeric values. Its public methods may include operations like ‘Plus’ (+), ‘Minus’ (-), ‘Assign’ and ‘Get’. This class can then be used, for example, to create instances such as ‘25’ and ‘31’ (which we assume will carry in their respective private memories representations of the values 25 and 31). We can then form the message:

\[ \text{Plus}(25, 31) \]

which is a message directed to the instance ‘25’, selecting the method ‘Plus’ and providing the object ‘31’ as a parameter. The method ‘Plus’, as the name suggests, will return an object that holds the sum of the values 25 and 31 (let’s call this object ‘56’). Note that it doesn’t matter, in fact, if the message was directed to the instance ‘31’ and providing ‘25’ as the parameter - the result will be the same, ie. ‘Plus’, in this case is associative like the standard arithmetic operator ‘+'.

Such system objects may of course be re-used by user-defined ADOs or other system objects. So, for example, we may have another numeric object called ‘Age’, and messages such as the following can be constructed:

\[ \text{Assign}( \text{Age}, \text{Plus}(25, 31) ) \]

In an object-oriented database system, there are also system classes to structure data. Such classes are generically referred to as Data Structures. They will have other objects in their private memories (members of the structure) and typically provide public interfaces to insert new members, remove existing members and also to broadcast messages to all current members.

Let’s say, for example, that there is a data structure class called ‘Collection’, with methods ‘Insert’, ‘Remove’ and ‘Broadcast’. We may then create an instance of this class, say ‘Col-1’, and populate its private memory with other objects by sending it messages of the form

\[ \text{Insert}( \text{Col-1}, \text{Obj} ) \]

where ‘Obj’ is some object identity. We can also remove existing objects in ‘Col-1’ using messages like

\[ \text{Remove}( \text{Col-1}, \text{Obj} ) \]
provided ‘Obj’ is a member of ‘Col-1’. And since every member of ‘Col-1’ is an ADO itself, we may of course direct messages to individual members, or using the ‘Broadcast’ method provided, direct a message to all members simultaneously, eg.

Broadcast( Col-1, Plus( 1 ) )

This particular message assumes that every member is a numeric object. The ‘Broadcast’ method will, for each member \( m \) in the collection, issue the message “Plus( \( m \), 1)”.

This facility is useful when some operation is to be applied to every object in the collection (it can be seen as a special case of repetition constructs in conventional programming languages, such as for loops). Of course, the message to broadcast can be any message to which members can respond.

The method ‘NEW’, in addition to creating a new instance of a data class, may also initialise the private memory of the instance. The initial values, if any, are themselves ADOs and are included in the message to the data class. Thus, a fuller form of the ‘NEW’ message structure is:

NEW( <data class>, <var>: <value>, …, <var>: <value> )

where <var> is an instance variable name defined in <data class>, and <value> is an object identifier.

There are basically two ways objects may be specified in the ‘NEW’ message. Suppose, for example, there is a data class called ‘Person’, and that there is already a database of existing objects with identities as shown in Figure 2-3.

![Figure 2-3 Example Data Class](image)

Then an object identity may be specified in the <value> part of the message, for example:

NEW( Person, Affiliation: DCS )

This will result in the creation of an instance with the variable ‘Affiliation’ set to the object named ‘DCS’.
A second method of specifying initial object values comes from noting that the ‘NEW’ message itself returns an object. Thus, the <value> parts of a message can be ‘NEW’ messages!

For example:

```plaintext
NEW(Person,
    Name: NEW(String, “Nick”),
    Age: NEW(Numeric, “40”),
    Affiliation: DCS
)
```

We assume that ‘String’ and ‘Numeric’ are data classes that are already defined. This message will thus initialise the ‘Name’ variable with a new ‘String’ object and the ‘Age’ variable with a ‘Numeric’ object. (Note that the syntax used here is informal. It is not our intention to describe any particular object-oriented system or its specific language constructions. We are more concerned with describing concepts and features of object-oriented systems in general. We frequently turn therefore to fairly intuitive graphical notations).

The two ways of specifying objects can of course be combined in one message, as illustrated by the last example.

**Class, State and Identity**

A class defines a *type* of objects, distinguished from other types by its particular memory structure and methods. An object of a class takes on the memory structure it describes, but is otherwise free to set values in its memory independently of other objects. The values of the instance variables of objects therefore constitute the *state* of the object. Thus two objects may be of the same type (instances of the same class) but may possess different states.

The ‘NEW’ method of a class, in addition to creating an object of the class, assigns a unique identity to it (conceptually, there is an infinite set of identities that the system can choose from!). This identity is *permanent*, in contrast to the state of the object which can change arbitrarily. Thus, Nick’s affiliation may change in the above example, but his identity remains unchanged. This formalises the notion of object reference mentioned earlier (similar to the notion of a pointer in conventional languages). Without identity, it would not be possible to refer to objects independently of their state, and object re-use would be impossible, ie. it would not be possible to have the *same* object as the value of an instance variable in more than one object.

The distinction between identity and state, however, does not apply to objects of so-called Base Classes. These are system classes of values that are atomic, such as integers, floating-point numbers, characters, etc. Such base objects do not have memory and thus cannot have a state that can vary independently of its identity. *Its identity is its value!*

Thus an integer ‘8’ object denotes both the object’s identity and its value. The state of a user-defined object will eventually be constructed from such base objects.
2.3 Definition of Private Memory

To define a data class, we must define

- A unique name for the class,
- A structure for the private memory of the class, and
- A collection of methods shared by all instances of the class

Private memory structure is defined as a binding of instance variable names to existing data classes. The idea of a ‘binding’ here is not unlike the idea of typing in (typed) programming languages. For example, variables in such languages are declared using constructs such as

```plaintext
<variable name> : <type>
```

e.g.,

```plaintext
x: integer;
y: real; …etc
```

The meaning of such a declaration is that the named variable is constrained to hold only values from the specified type. Likewise, a binding is an association of an instance variable name and a data class, using declaration constructs such as

```plaintext
<instance variable name> : <data class name>
```

As a class is basically a type, the meaning of such a binding is that the instance variable is constrained to hold only instances from the specified data class. Such a binding is thus also referred to as a Variable - Domain binding.

Suppose that ‘String’, ‘Numeric’ and ‘Collection’ are system classes. Then the following is a definition of a class and its private memory structure (again, the syntax is notional):

```plaintext
class Firm;

instance variables
Name: String;
Employees: Collection;

end Firm;
```

The class ‘Firm’ in turn can be the domain of instance variables in some other class, for example:

```plaintext
class Person;

instance variables
Name: String;
Age: Numeric;
Affiliation: Firm;

end Person;
```
Bindings therefore define *associations between data classes* which can be graphically depicted in Object-Oriented Data Structure Diagrams. The classes and relationships introduced by the above definitions is shown in Figure 2-4.

![Figure 2-4 Object-Oriented Data Structure Diagram](image)

### 2.4 Definition of Methods

A method is defined by specifying

- a Message Template, and
- a Method Body

A Message Template defines the structure of messages to which the method will respond. It must include

a) a name to match the selector of an incoming message;
b) a specification of a target instance
c) a specification of formal parameters in the form of a list of Variable–Domain bindings.

The above of course mirrors the structure of messages presented earlier. Variables in the formal parameters part of the template are names that can be used in the message body (they are different from and should not be confused with the instance variables of the class). The domain parts of the bindings must be names of existing classes. Such a formal parameter list will match the parameters of an incoming message if each parameter value is from the corresponding domain in the formal parameter list. The corresponding formal variable will then be bound to that value during execution of the message body.

Suppose for the class ‘Firm’ above, we wish to define a method to respond to a message to add a new employee. The following then is a possible template:

New–Employee   (  T: Firm,   E: Person )

- This specifies the input parameter, which is an instance of ‘Person’
- Name chosen for the message selector

*This specifies the target*
With this template, a message such as

\[
\text{New-Employee ( X, Y )}
\]

will be recognised and accepted by the instance X, if X is an instance of ‘Firm’ and if Y is an instance of ‘Person’. In such a case, the formal variables T and E will be bound to X and Y respectively in the execution of the message body. Any other message structure will be rejected.

If the method also returns a value in response, the template will also specify this by writing a Variable-Domain binding after the formal parameters. For example:

\[
\text{New-Employee ( T: Firm, E: Person ) R: Logical}
\]

The binding “R: Logical” specifies the response the method will generate. We assume ‘Logical’ is a base system class with \text{true} and \text{false} as its only possible instances. Thus the response R will be either \text{true} or \text{false}.

The body of a method is specified as a number of message expressions involving the formal variables in the template and instance variables of the class. If a result is specified in the template, some expression must also cause the result variable to be bound to a value. Thus, for the above example, the following might be the method body:

\[
\text{Set ( R, Insert ( T.Employees, E ) )}
\]

This method body may be interpreted as follows:

- First, issue the message “Insert ( T.Employees, E )”. The target of this message is the instance found in the instance variable ‘Employees’ of T, which at this time is bound to the very instance that is responding to the original message. The parameter of the message is the formal variable E which at this time is bound to the instance of ‘Person’ in the original message.

- Assume that the response to the ‘Insert’ message is a logical value. That is, the method body expression is reduced to the following when the ‘Insert’ message responds with a value, say V:

\[
\text{Set ( R, V )}
\]

This is also a message that is sent to the variable R to set its value to V. The net effect therefore is to generate a response containing V to the sender of the ‘New-Employee’ message.

As another example, consider the definition of the method ‘Print’ for the class ‘Firm’:

\[
\begin{align*}
\text{Message Template:} \\
\text{Print ( T: Firm ) R: Logical}
\end{align*}
\]

\[
\text{Method Body:}
\]

\[
\begin{align*}
\text{Set ( R, Print ( T.Employees ) )}
\end{align*}
\]
Print ( T.Name );
Broadcast ( T.Employees, Print );
Set ( R, true )

This example underlines the object-oriented style of processing, which is based on message passing. The intention of this method is to print the name of the ‘Firm’ instance and all its employees. But as the latter are ADOs themselves, the processing at this level cannot directly print to printer their encapsulated information, since such information is hidden. The options at this point therefore are to collect from the objects the relevant information for printing, or to pass the responsibility of printing to the objects themselves. In both cases, messages must be sent to the objects, and the objects must of course have corresponding methods to either return requested information or to perform the printing themselves. The example above assumes that instances of ‘String’ and ‘Person’ can handle printing and thus the ‘Print’ message is passed to them.

2.5 Summary

A data class is a description of ADOs all having the same private memory structure and public interfaces (methods). Each ADO in an object-oriented system belongs to a particular data class and is created as an instance of that class.

A data class is defined by specifying its private memory structure and its methods. The definition of its private memory is essentially a set of bindings of instance variables to existing data classes. Methods are typically defined by messages that are sent to other objects in the system. The definition of a data class is called a Class Definition Expression.

We distinguish between system data classes on the one hand, and user-defined data classes on the other. The former are a part of a given object-oriented (database management) system. The latter are the results of class definition expressions, drawing on system classes and other user-defined classes, that collectively describes the behaviour (data structures and operations) of a particular database system.

Figure 2-5 Object-Oriented Database Schema.

A set of data classes therefore defines all possible ADOs that can populate the system. In the context of a database system, such a set of data classes therefore forms an Object-Oriented Database Schema. This is illustrated in the figure above. Note that such a
Object-Oriented Data Model

schema not only defines the data structures but also purpose-oriented behaviour of the system (ie. operations that can be applied to data objects).
3 Dynamic Binding and User Interface

3.1 Introduction

ADOs combine data structures and procedures operating on them. Messages serve both the purpose of procedure activation and of data passing. An object-oriented database is a collection of ADOs and a user interacts with such a database basically by sending messages to activate processing and receiving data passed back in response. This is depicted in Figure 3-1.

![Figure 3-1 A view of an object-oriented DBMS](image)

We have seen that data classes define the meta-structure of an object-oriented database as well as the behaviour of ADOs that instantiate them. Different applications will typically call for different data classes to be defined, i.e. the data objects and their associated operations are application dependent. It is not possible, therefore, to have a standard user interface of generalised operations that is application independent. Instead, the user interface for an application must be defined in the object-oriented database schema itself. To facilitate this definition, there are special system data classes that provide some basic building blocks.

3.2 User Interface and System Data Classes

Recall that every object-oriented system includes a number of predefined system classes. A system class is one of

- a Base Class, such as ‘Integer’, providing base objects upon which the states of objects are eventually constructed
- a primitive Data Item, such as ‘String’ or ‘Numeric’; unlike base class objects, these have states that can inspected or changed
- a Data Structure, such as ‘Collection’, ‘Array’, ‘Queue’, ‘Stack’, etc, used to combine other data objects into entities with predefined properties
- a User Interface, used to define application-specific user interfaces (in effect, a data manipulation language)
We have seen examples of the first two types earlier and how they are used, and in fact essential, as components of user-defined classes. A user interface system class differs in that they are designed to interact with a user, by visualising data objects or by reacting to user actions or both. Given today’s sophisticated visual presentation and interactive input devices, user interface system classes typically include a set of templates for constructing graphical, direct-manipulation interfaces that are visualised on computer screens and provide for interactive user actions through various devices such as a keyboard, a mouse, a touch pad, a touch screen, etc.

For example, it is quite common to find a system class called “Push-Buttons”, or something similar. The class represents objects that when visualised presents to the user what looks like a button on the screen, which he/she can then manipulate using interactive input devices like a mouse. We illustrate this in the figure below. As shown, the ‘NEW’ message brings a number of other objects as parameters. The Push-Button class is assumed to respond by first creating for each parameter an on-screen 3D-look button, with each button labelled with text obtained from the input parameter objects. The method then waits for the user to select one of the buttons, upon which the corresponding object will be returned as the final response.

![Figure 3-2](image)

**Figure 3-2** An example of creating and visualising user interface objects

This, of course, is neither a complete nor definitive description of interface classes like Push-Button. It is only intended to outline and highlight that their effects and capability are largely to do with human-computer interaction, viz. visualising information for the user and acquiring information from the user.

The user interface objects created, such as the buttons above, are destroyed once a selection is made. This is typical of the nature of interface objects - they are discarded once they have served their purpose. Interface objects, therefore, have a transitory existence compared to the more persistent data objects in the database.

![Figure 3-3](image)

**Figure 3-3** Editable fields example
Interface objects, however, may cause instances of other data classes to be created in the course of processing. This is illustrated in Figure 3-3. Here we have an ‘Input-Field’ interface class that, upon receipt of a (parameterless) ‘NEW’ message, creates an input field object that is visualised on the screen as an edit box. The field object waits for the user to finish typing text in the box, at which point the text is returned as an instance of ‘String’, say, while the field object itself is discarded. Note that since the response can be an object, messages to interface objects can be the target of other messages or be passed as parameters in other messages.

There are also system classes used principally for their ‘side-effects’, ie. their responses to messages are not so much to be found in the objects returned but in changes they cause to the environment, eg. printing information, sending electronic messages, saving information outside the database, etc. Figure 3-4 shows an interface object designed for printing. The response it generates, an ‘OK’ signal in this case, is largely inconsequential and intended only to inform the message sender of the status of the task.

![Figure 3-4 System objects with ‘side-effects’](image)

We should also note at this point that many system classes, not just user interface classes, can cause side-effects. Printing, especially, is a capability of most system objects, including data structures. That is, their public interfaces include a predefined ‘Print’ method that when activated will cause information they contain to be printed in some predefined format.

For example, the message

```
Print( New( String, "Welcome") )
```

will first create a new string instance containing the text string “Welcome”. This new instance will then respond to the ‘Print’ message, causing “Welcome” to be printed and passing status data back to the message sender.

### 3.3 Dynamic Binding

In conventional programming languages, the name of a procedure statically defines the code to be executed, ie. it is possible to determine at compile time the code to be activated for any procedure call. Suppose for example, the following procedure was defined:
Object-Oriented Data Model

\[ \text{Print\_Person( Name: String, Age: Numeric)} \]
\[ \{ \ldots \text{code--for--Print\_Person} \ldots \} \]

The procedure name ‘Print\_Person’ will be statically bound to \{\ldots \text{code--for--Print\_Person} \ldots\}.

A call such as

\[ \text{Print\_Person( “Nick”, 40 )} \]

will cause a procedure ‘jump’ to \{\ldots \text{code--for--Print\_Person} \ldots\}, after the procedure’s formal parameters Name and Age have been bound to the actual parameters “Nick” and 40 respectively. Any call to ‘Print\_Person’ will in fact activate the same code, differing only in the actual parameters passed to the formal parameters. This is true even for block-structured languages that allow names to be re-used for procedures defined in inner blocks, since scoping rules in such languages allow references to names to be resolved at compile time.

In object-oriented systems, messages play the role of procedure calls and method selectors are analogous to procedure names. In contrast, however, it is not possible to determine statically the code to be executed for a given message. This is because the instance to which a message is directed can only be determined dynamically (at run-time) and the code associated with the specified selector is likely to be different for objects from different classes.

Suppose for example, there were two classes ‘Employee’ and ‘Student’, each with the method ‘Print’ defined in its public interface. Suppose further that “Nick” and “Alex” are instances of ‘Employee’ and ‘Student’ respectively.

The messages:

\[ \text{Print( “Nick” ), and} \]
\[ \text{Print( “Alex” )} \]

while having the same method selector, will execute different code - that defined in the class ‘Employee’ for the former, and ‘Student’ for the latter.

The reader may have noted that the class of an instance expression can be statically determined, eg. a message like ‘New( <class>, \ldots )’ clearly identifies the class and thus any message with this expression as its target determines the methods (and code) to be activated. Furthermore, instance variables are bound to data classes, and a message template specifies the data class of its response, again suggesting that we can tell the classes of objects we are dealing with statically. This is true, of course, but object-oriented systems also support inheritance and polymorphism. That is, the actual objects may be instances of derived classes that redefine code for some or all of the methods inherited. It is not possible therefore to determine the method to invoke until run-time.

We will treat inheritance in the next chapter, but for now, we will simply re-assert that the binding of a method name to execution code is dynamic.

The dynamic binding property of object-oriented systems increases their flexibility enormously. For example, the definition of a method in a class does not become overly dependent on the domain bindings of instance variables, as long as the specified domains offer (at least) the same public interfaces used in the method body. This is illustrated in
Figure 3-5 which shows the definition of a ‘Print’ method for the class ‘Bank-Account’. Note that the body of the definition sends a ‘Print’ message in turn to the objects in the variables ‘Owner’ and ‘Sum_Available’. As long as these objects can respond to such a message, it doesn’t matter what classes they belong to, i.e. we can change the domain binding of the variables without having to change the method definition! Eg. the domain of ‘Owner’ can be changed from ‘Student’ to ‘Employee’, and that of ‘Sum_Available’ from ‘Dollar’ to ‘Numeric’, and the ‘Print’ method will still execute correctly provided that ‘Employee’ and ‘Numeric’ can respond to ‘Print’ messages. Such object transparency, afforded mainly through public interfaces and dynamic binding, plays a very important role in object-oriented systems.

![Object transparency through dynamic binding](image)

**Figure 3-5** Object transparency through dynamic binding

### 3.4 Summary

Data associations in object-oriented databases are based on the re-use of objects in the private memory of other objects, i.e. on the binding of instance variables to data classes. This is in contrast to the relational data model which manifest relationships through foreign keys in relations.

The top half of Figure 3-6 shows the familiar example of a relational schema we have used in earlier chapters. Note that the relationship between the entities ‘Customer’, ‘Product’ and ‘Transaction’ are captured in the latter through the shared attributes ‘C#’ and ‘P#’. These attributes will take on, respectively, values from the corresponding attribute in ‘Customer’ and ‘Product’, serving therefore as references (or pointers) to the relevant tuples in those relations.

In contrast, the figure also shows in the bottom half a simple and direct translation from relation definitions to data class definitions - each relation corresponds to a data class and the relation’s attributes are represented by instance variables. An instance of a data class then corresponds to a tuple of the corresponding relation. Note, however, that such instances are directly re-used in the private memory of ‘Transaction’ instances instead of being referenced through keys (there is thus no need for the attributes ‘C#’ and ‘P#’ in the object-oriented model and so we omit them in our translation from the relational model).
The object-oriented approach to databases offers the following advantages:

1. Level of granularity of data objects:

Methods may be defined for one data class independently of other data classes and of the overall database structure. Dynamic binding of methods provides capability (such as printing) that is object-transparent and that is resilient to changes in object relationships (such as changing variable–domain relations).

2. Re-use of data objects:

The private memories of instances allow the expression of arbitrarily complex associations among them, directly and naturally. The relational model, in contrast, often force the invention of arbitrary key attributes and values to allow relationships to be expressed through foreign keys.

3. Purpose-oriented behaviour of data objects:

Data classes may be designed to reflect the domain entities and their roles or purposes. In a real business, for example, transaction records allow us to compute the profit/loss of each transaction, or to generate invoices. These purpose-orientation is quite simply modelled by methods. The user interface, in particular, will reflect the purpose of the application at hand.
4 Static Inheritance

4.1 Introduction

Thus far, we have seen that data classes define an object-oriented schema. Such data classes are described by users using class definition expressions. A user-defined class will typically use system-provided classes (and other defined classes) to structure its private memory and define its methods. Memory is structured as a set of variable-domain bindings, allowing essentially client-server associations between ADOs, ie. an instance (the client) can call on the services of objects (the servers) that are in its private memory. The resultant database schema, and its instantiations, is therefore a ‘flat’ collection of objects in that all objects have equal stature and can call on any other object for services published in their public interfaces.

Many applications, however, will have entities that have similar private properties, ie. sharing many private memory structures and public methods. If the only means of structuring available were variable-domain bindings, then the description of such entities as data classes will see many similar definitions with possibly many duplicated parts.

For example, consider the data classes in Figure 4-1. All three classes have in common the instance variables ‘Name’ and ‘Affiliation’, and the methods ‘Get-Name’, ‘Affiliation’ and ‘Print’. But each also has a unique instance variable and method. Given class definition mechanisms discussed so far, these classes will have to be defined separately with the common components duplicated in each definition. This is wasteful and some means of sharing common definitions while still allowing for differences is clearly desirable.

![Figure 4-1](image)

4.2 Static Inheritance of Properties

Such a means is in fact found in all object-oriented systems and is called static inheritance. The idea is that a new data class can be defined to inherit from an existing class all of the latter’s private memory structure and public methods.

Consider for example, the situation in Figure 15-2.
Assume that the class `Person` has fully defined its instance variables and public methods. We can extend the class definition expression introduced earlier to include an inheritance specification, e.g.

```java
class Employee inherits Person; ...
```

or, equivalently, depict such a definition pictorially as shown in the figure. Then all properties defined in ‘Person’ becomes also properties of ‘Employee’, even though the latter’s definition makes no explicit mention of instance variables or public methods. That is, if we created an instance of ‘Employee’, that instance can respond to ‘Get_Name’ and ‘Get_Affiliation’ messages.

We say in cases like this that ‘Person’ is a superclass and ‘Employee’ is a subclass. Of course, there can be many subclasses of a given superclass. For example, the classes ‘Student’ and ‘Author’ in Figure 4-1 can also be subclasses of ‘Person’.

Subclass definitions therefore achieve the desired sharing of common properties. But if this is all we can do in subclass definitions, a subclass would amount to nothing more than a synonym for its superclass. Thus, in addition to inheriting superclass properties, a subclass may additionally define new properties, i.e. new instance variables and/or methods. This is illustrated in Figure 4-2, which shows the definition of the three classes in Figure 4-1 as subclasses of the class ‘Person’. Note that the inherited properties do not have to be duplicated in the subclasses. Instead, each subclass need only define properties relevant to itself, thus differentiating itself from the superclass and from sibling subclasses.
Subclassing may thus be seen as *specialising* a superclass to a subset of objects that satisfy added properties. But besides adding new properties, specialisation can also involve *suppressing* and *redefining* (or *overriding*) selected properties.

Suppression discards specified properties. Say, for example, that we introduce a subclass for people who have retired and call it ‘Retired_Person’ (Figure 4-3). A retired person will have no affiliation and the method ‘Get_Affiliation’ and the instance variable ‘Affiliation’ are thus irrelevant, i.e. we wish to inherit all properties except these. In the subclass definition, therefore, we explicitly suppress these properties. Instances of ‘Retired_Person’ will consequently not have ‘Affiliation’ as a variable nor can they respond to ‘Get_Affiliation’ messages. They otherwise would behave like instances of ‘Person’.

There are also situations when an inherited property is not suppressed but is modified instead, i.e. the names of the properties are retained but their attributes are changed. Thus, inherited instance variables may be bound to a different domain, and inherited methods may be assigned different method templates.

Figure 4-4 shows a situation when overriding the domain binding of inherited variables makes sense. For example, a student account is a bank account such that the account owner is a student, and a corporate account is also a bank account but the account owner must be a firm (company or corporate body). Each subclass therefore redefines the domain binding of the ‘Owner’ variable to appropriate domains. Note that while we have redefined some variable-domain bindings, the inherited methods remain unchanged,
assuming that the public interfaces of the new domains offer at least the same methods as the overridden domains (recall our discussion in the last chapter of object transparency due to dynamic binding).

![Figure 4-4 Redefining/Overriding inherited variable-domain bindings](image)

If we wished, inherited methods can be redefined too. And we may choose to redefine a method in one subclass but not in another. For example, the ‘Print’ method above may be redefined for the ‘Corporate_Account’ subclass but not for the ‘Student_Account’ subclass (see Figure 4-5).

![Figure 4-5 Redefining/Overriding inherited methods](image)

Any class can be the superclass for subclass definitions, including system-provided classes. Furthermore, subclassing can be continued to arbitrary levels, ie. a subclass may itself be the superclass of other classes. For example, the ‘Employee’ subclass in Figure 4-2 may be used as the superclass of two new subclasses, say, ‘Permanent’ and ‘Temporary’. Continuing in this way, an inheritance hierarchy of any depth can be constructed. Static inheritance therefore introduces a hierachical structure to what otherwise would be a flat database schema.
An instance of any class in this hierarchy will have all the properties of that class, including all new properties it defines and all the unsuppressed and redefined properties it inherits from its superclasses.

It is important to realise that while two classes may be related through inheritance, their instances are separate ADOs. Thus, if B is a subclass of A, and I_B and I_A are their respective instances, the private memories of I_B and I_A are unrelated. That is, while both can have an instance variable named V, the value of V in I_B is independent of the value of V in I_A. In other words, inheritance is a device for sharing descriptions of properties among data classes - not the sharing of those properties among their instances.

### 4.3 Abstract Data Classes

With inheritance defined, a data class can serve two different purposes:

1. As a template, for the creation of ADOs that will populate a database, and
2. As a superclass describing common properties of a number of subclasses

Any data class may in fact serve both purposes within an application. However, there are situations where a data class is created solely for the second purpose above, with no intention of ever creating their instances, or that their instances would be incomplete entities and thus meaningless in the application context. Instead, instances are created only from their subclasses which presumably will describe additional properties that make them meaningful. Such data classes will be referred to as Abstract Data Classes.

For example, an abstract bank account can be described as in Figure 4-6. It defines a number of methods and instance variables, but the variables are not as yet bound to any domain. Such incomplete descriptions are allowed in the case of abstract data classes—the intention is for subclasses to specialise it and fill in the gaps. As such, it does not make sense to directly create instances from it.

![Figure 4-6 An Abstract Bank Account Class](image)

From such an abstract class, we may derive more complete subclasses, as in:

```plaintext
class Corporate_Account inherits Bank_Account;

instance variables  Domain bindings specified
                      for the variables
```
Object-Oriented Data Model

Owner: Firm;
Sum_Available: Dollar;
end Corporate_Account;

The missing domains are filled in by this definition (methods are inherited unchanged). This then forms a complete class definition and instances can therefore be created from it.

Recall that a data class represents a set of objects (ie. ADOs) all having the properties and behaviour it describes. A subclass in fact denotes a *subset* of the objects described by its superclass. In other words, an object that is a member of the subclass is also a member of the superclass. We should clarify at this point the distinction between being ‘an instance of’ and being ‘a member of’ a data class. When we say that an object is ‘an instance of’ some class, that object was created by sending a ‘New’ message to that class. Obviously, it is also ‘a member of’ that class. Additionally, however, if the class has a superclass, the object is also ‘a member of’ the superclass, even though it is not ‘an instance of’ the superclass.

Class membership, rather than instantiation, is the basis for type checking (eg. parameters in messages, values to be assigned to instance variables, etc). In the ‘Print’ method definition above, for example, the expected parameter is a ‘Bank_Account’ object. This is actually interpreted to mean any object that is ‘a member of’ the ‘Bank_Account’ data class. Thus, any instance of a class that directly or indirectly inherits from ‘Bank_Account’ is a valid parameter. This is why, when we defined ‘Corporate_Account’, we do not need to modify the method template.

Abstract data classes play a very important role in object-oriented systems. Many system data classes, for example, are really defined as abstract classes. Static inheritance then allows users to derive from these abstract classes new application-specific classes. The idea of persistent database objects is in fact abstracted to a special abstract class called ‘Database_Object’, and all other classes describing database objects inherit properties from it, such as to store/delete an object into/from the database. This is illustrated in Figure 4-7.
4.4 Definition of an Object-Oriented DBMS

We can now define more precisely what an object-oriented DBMS is. Specifically, it is one that:

1. provides a number of system data classes designed for database creation; these include classes to define persistent database objects, to create data structures, and to create application-specific user interfaces for data manipulation
2. supports static inheritance and class definition expressions to create user-defined application-specific data classes
3. allows users to create, modify and access a database through predefined methods of the system data classes

4.5 Summary

Static inheritance allows us to define a new data class N (the subclass) from an existing data class C (the superclass) by:

a) adopting for N all the properties of C (ie. C’s memory structure and methods), then
b) (optionally) suppressing in N some of the adopted properties, then
c) (optionally) adapting (redefining or overriding) in N some of the adopted properties, and finally
d) (optionally) adding in N new properties not found in C

Typically, the definition of N will include some suppression or adaptation or addition of properties (although they are optional) - otherwise N would only be a synonym for C.

Static inheritance is basically a descriptive device, allowing several classes to share the same property descriptions. Many applications benefit from this since they usually involve many similar entities that can be organised in an inheritance hierarchy. Static inheritance facilitates a concise and natural definition of such entities as data classes.

Some data classes are used only as superclasses, ie. no instances are ever created from them. Such classes are termed Abstract Data Classes. They play an important role in object-oriented systems and many system data classes are abstract. In particular, the abstract class ‘Database_Object’ defines many basic properties needed by persistent database objects. Users define application-specific data classes by inheriting properties from abstract data classes provided by an object-oriented DBMS.

We have largely used, and will continue to use, an informal graphical notation to express object-oriented concepts. This is because our focus is more on the general properties of object-oriented database systems rather than any specific formal system. The main notations for static inheritance are summarised in Figure 4-8.
Figure 4-8 Summary of graphical notation for static inheritance
5 Dynamic and Multiple Inheritance

5.1 Introduction

Static inheritance (Chapter 15) is a descriptive device to define new data classes from existing data classes. Recall that a data class is defined by describing its private memory structure and methods (Chapter 13). With static inheritance, part of this description can be achieved by a combination of adopting, adapting and suppressing the memory structure and methods of an existing, similar data class - the superclass. The resultant description may also be extended with definitions of new memory structures and methods not available in the superclass.

Static inheritance extends the class definition apparatus and is orthogonal to variable-domain binding specifications of private memory structure. The latter in fact describes a relationship between instances of data classes. That is, if a data class C binds a domain S (another data class) to one of its instance variables V, it really says that any member of C can take any member of S as the value for V. Different members of C of course have their own private copy of V. In contrast, static inheritance expresses a relationship between data classes only, not between the private memories of their instances. That is, if C were the superclass of S, an instance of S does not inherit memory values from an instance of C! Their private memories are in fact independent of each other.

In many database applications, however, inheritance of memory values of one instance by another is a useful feature. Consider, for example, the database of departments and their employees. Let’s assume there can be an arbitrary number of departments and each department can have an arbitrary number of employees. Each employee, however, can belong to only one department. To model this in object-oriented terms, we define two data classes: ‘Department’ and ‘Employee’. ‘Department’ has the variables ‘Title’ and ‘D_Phone’ to hold respectively the name of the department and its general line phone number. ‘Employee’ has the variables ‘Name’ and ‘Phone’ to hold respectively an employee’s name and personal extension, if any. This is illustrated in Figure 16-1 (a), which also shows some associated methods of each class.

The database itself will therefore be clusters of instances wherein each cluster will have one ‘Department’ instance and one or more ‘Employee’ instances that belong to it (Figure 16-1(b)). Clearly, there is a relationship between an instance of employee and the instance of department. We should, for example, be able to get the department name given an employee instance. Furthermore, if the department name were to change, that change should be reflected in all associated employee instances. That is, we would like employee instances to inherit the value of ‘Title’ from the department instance. Static inheritance cannot do this (thus the ‘?’ in the relationship drawn in Figure 16-1).

\[1\] Note that we use ‘member of’ rather than ‘instance of’ since this more accurately describes the semantics of variable-domain bindings (in a system that supports static inheritance and abstract data classes). Of course, the member of C, if not also an instance of C, must not have suppressed the variable V.
We can actually represent the required relationship above using variable-domain bindings, as follows. Define in the ‘Employee’ data class a variable called ‘Dept’ and bind it to the domain ‘Department’ (see Figure 5-2). The idea is that every instance of employee will re-use the instance of department they belong to as the value of their ‘Dept’ variable. To get the department name from an employee instance, we must also define a method that relinquishes the task to the instance held in the ‘Dept’ variable. Another method will also be needed to set the ‘Dept’ variable, say, ‘Set-Dept’ (not shown in the figure). Note that in this solution to the problem, adding an employee to a department is a two stage process: first, create a new employee instance, then send it a message to set its department to a selected instance of department (assuming it already exists in the database).

The solution using variable-domain bindings requires explicit ‘programming’ on the part of the developer. Alternatively, dynamic inheritance provides a way of implicitly establishing instance value and method inheritance among data class instances.
5.2 Dynamic Inheritance

Dynamic inheritance among instances of data classes are still specified at the level of data class definitions. To distinguish between static and dynamic inheritance, we will use the graphical notation as shown in Figure 5-3 to denote the latter (ie. double lines and borders as opposed to single lines and borders for static inheritance).

When a data class S dynamically inherits from a data class C:

a) Every instance of S will be associated with one, but not necessarily the same, instance of C. Effectively this describes the cluster organisation as shown in Figure 5-1

b) An instance of S will inherit all the private memory structure and contents of the C instance it associates with, unless explicitly suppressed or redefined

c) An instance of S will inherit all the methods defined in C, unless explicitly suppressed or redefined (as per static inheritance)

Thus, the definition in Figure 5-3 admits clusters of instances each of which contains a department instance and one or more employee instances. Each employee instance in a cluster will inherit the memory structure and contents of the department instance. Furthermore, each also inherits the methods defined in the ‘Department’ data class, except for ‘Set-Phone’ and ‘Get-Phone’ which are explicitly redefined in the ‘Employee’ data class.

Inheritance of the memory structure and content is best interpreted as sharing, rather than copying. Thus, if a department instance is sent a ‘Set-Title’ message, the new value of its ‘Title’ variable will be instantly available to all employee instances associated with it. In a similar fashion, inheritance of methods is best thought of as an indirection, ie. if a message to an inherited method is received, it is resent to the “super-instance”. Thus we can send a ‘Get-Title’ message to an employee instance and receive in response the value in the ‘Title’ variable of the associated department instance. Note that we can also send a ‘Set-Title’ message to an employee instance—the effect will be equivalent to directly sending the message to the associated department instance!

In effect, the super-instance becomes a server of values and their manipulation (through methods) for the sub-instances which constitute its clients. This is similar in principle to the client-server relation established through variable-domain bindings. The difference,
however, is that relations through variable-domain bindings must be explicitly installed
and managed, ie. methods must be written to install a server in the client’s memory and to
resend messages to it.

In contrast, the client-server relation under dynamic inheritance is established at client
creation time. More specifically, creating an instance inheritance hierarchy involves the
following steps:

a) create an instance of the superclass, ie. the server, by the normal mechanism of
   sending a ‘NEW’ message to the corresponding data class

b) create an instance of a client by sending the ‘NEW’ message to the server!

Step (b) may be repeated to install as many clients as required for any given server. Having
done this, the system will automatically manage all such client-server
communication. Note that in step (b) above, the ‘NEW’ message is sent to an instance
rather than a data class. This is only defined when dynamic inheritance has been
specified. That is, the ‘NEW’ method of a subclass of a dynamic inheritance definition is
relegated to instances of the superclass (however, see next section).

Of course, client-server behaviour does not apply to overridden properties. Thus, the
message to ‘Set-phone’ sent to an employee instance will change that instance’s memory,
leaving the server memory unchanged. Likewise, ‘Get-phone’ sent to an employee
instance will return the value of ‘Phone’ rather than ‘D_Phone’. However, if new or
overridden client methods deal with inherited instance variables, messages addressed to
such variables will be resent to the server.

In object-oriented systems that support both variable-domain bindings and dynamic
inheritance, the user of course has a choice over which method to use to address
situations exemplified by the department-employee database. Variable-domain bindings
are more general, allowing arbitrary configurations or organisations of instances.
Dynamic inheritance forces a particular configuration of instances, but where such
configurations are intended, it is more convenient and natural.

### 5.3 Multiple Inheritance

Simply put, multiple inheritance is the inheritance of properties from two or more
superclasses (see Figure 5-3). In the case of static inheritance, the effect of multiple
inheritance is the union of properties of the superclasses. Thus, in Figure 5-3, class C will
have variables X and Y and methods 1 and 2 defined for its instances.
There are obviously conditions for this to be a well-defined subclass definition. First, if a variable name appears in more than one superclass, the domain it is bound to in all its occurrences should be the same. Even then, different methods that manipulate the variable must be examined to see that they would not interfere with one another, since in the subclass there will be only one copy of the variable. On the other hand, if the variable is bound to different domains in different superclasses, it suggests they serve different purposes and often renaming the variables will remove the problem (of course, occurrences of the variable in method bodies must also be amended). Second, if a method name occurs in more than one superclass, their definitions must be examined. Even if their definitions are identical, we must ascertain that the variables they use serve the same purpose over the different classes. If not, the variables and the method name should be renamed. On the other hand, if their definitions are different, then renaming the method name may be sufficient.

These are quite complex conditions and are usually difficult to ascertain. The simplest condition to guarantee well-definedness for static multiple inheritance is orthogonality of the superclasses, i.e. when they do not have common properties. For these reasons, static multiple inheritance is rarely used in practice.

Dynamic multiple inheritance, on the other hand, turns out to be quite useful and suffers less of the problems mentioned above. An example is shown in Figure 5-4. A ‘Person’ dynamically inherits from a ‘Firm’ and a ‘Soccer Team’. Intuitively, this models a database of persons and, for each person, the firm he/she is affiliated with and the soccer team he/she plays for. Effectively, such multiple dynamic inheritance sets up multiple servers for a given client, and messages sent to the client will be automatically rerouted to the appropriate server. Thus, the message ‘Get-Title’ sent to a ‘Person’ instance will be rerouted to and serviced by the ‘Firm’ server, whereas the ‘Get-Team’ message will be rerouted and serviced by the ‘Soccer Team’ server.
Of course, the server classes above are already orthogonal to one another and no ambiguities arise in handling incoming messages. Ambiguities will arise only if the different superclasses have methods of the same name. But this is easily circumvented by renaming. Common variable names in the superclasses do not cause problems since they exist in their own memory space, so long as methods that operate on them are not redefined and no new methods that refer to them are defined in the subclass.

There is a slight problem, though, with regard to installing multiple servers. The mechanism of relegating the ‘NEW’ method of the subclass to superclass instances is not appropriate as there is more than one superclass. Different object-oriented systems may differ in how they handle this, but the essential operation is creating a client object and binding it to one or more server objects. A uniform approach, therefore, is to retain the ‘NEW’ method with the subclass but to include in the message template parameters that represent the servers of the client object to be created.

5.4 Summary

Dynamic inheritance allows us to describe and establish run-time relationships between instances. The relationship is essentially that of a client (an instance of the subclass) and a server (an instance of the superclass). It differs from variable-domain relationships in that it describes a particular organisation of instances - essentially clusters of clients served by a server (or more, in the case of multiple inheritance) - and that client-server communication (through messages) is automatically managed by the system. Because it deals with instance relationships, it is sometimes also referred to as instance inheritance.

Dynamic inheritance and variable-domain specifications of instance relationships may be used in combination. The latter, while general enough to describe any configuration of instances, may be a little too cumbersome for situations that dynamic inheritance naturally models. The choice, however, is with the user.

Multiple inheritance - inheriting properties from more than one superclass - can be applied to both static and dynamic inheritance. The complex conditions of well-formedness for static multiple inheritance, however, makes its use rare in practice. Dynamic multiple inheritance, on the other hand, is useful, offering a means of setting up multiple servers for a client.
Object-Oriented Data Model
6 Object Identity and Database Query

6.1 Introduction

In chapter 15, we characterised an object-oriented DBMS as one that:

a) provides a number of system data classes designed for database creation; these include classes to define persistent database objects, to create data structures, and to create application-specific user interfaces for data manipulation

b) supports inheritance and class definition expressions to create user-defined application-specific data classes

c) allows users to create, modify and access a database through predefined methods of the system data classes

These primary functions are depicted in Figure 6-1, showing particularly a key system data class—‘Database Object’. All database objects derive from it, i.e. application-specific data classes defined by users inherit directly or indirectly from it, as do all other system classes including the data structure classes. All database objects therefore inherit from ‘Database Object’ some necessary methods for database manipulation, particularly the ‘Store’, ‘Delete’ and ‘Get’ methods.

![Figure 6-1 Object-oriented database creation and retrieval](image)

It is important to realise that when an object is created, it is not automatically inserted into a database. In other words, objects may be created and manipulated during a user session without ever inserting them into a database. Such objects will be transient, however, and are discarded at the end of a session (user interface objects are mainly of this type). To be part of a database, an object must be explicitly inserted. This is the function of the ‘Store’ method. Once inserted, objects will persist unless explicitly removed. And this is the role of the ‘Delete’ method.

The ‘Get’ method is for non-destructive retrieval of database objects, and thus central to database processing. Its message template is:
Object-Oriented Data Model

Get (Target: Object) Response: Object

‘Object’ in the template above is an abstract system class that is the superclass of all classes. In other words, any ADO we create will be a member of ‘Object’. Thus, the target of a ‘Get’ message can be any ADO whatsoever. Likewise, the response to a ‘Get’ message can also be any ADO.

The ‘Get’ message sent to a database object, returns the target object itself, which can then be manipulated in the usual way (eg. to print, to update its state, etc), and later restored into the database if necessary. This is illustrated in Figure 6-2. In the illustration, the target object is specified by its identity. But if we must always explicitly specify the target object identity, ‘Get’ would be quite uninteresting and somewhat limited in use. Database retrieval capabilities should also include implicit specifications of target objects, particularly involving *predicates over object states*. In other words, we should also be able to retrieve objects based on their (partial) content.

![Figure 6-2 Database object retrieval using the ‘Get’ message](image)

Remember that an object may be a data structure, ie. a collection of other objects. Thus it is also possible for the target and response to ‘Get’ to be a collection of objects. What we would really like is to send ‘Get’ to a collection (perhaps the entire database) and receive a response which is a smaller collection satisfying certain predicates over states of the target collection’s objects. If we can do this, we can cascade ‘Get’ messages such that the result of one becomes the target of another, until the desired object is retrieved. This more general query facility is the focus of this chapter.

6.2 Object Identity and Addressability

Let us re-examine the anatomy of a message. Each message comprises three components:

\[ S (T, P) \]

where S is a selector (ie. method name), T is the target object to which the message is directed, and P is zero or more parameters. So far in our discussion, we have assumed
that T is a name that uniquely identifies some object. In most object-oriented systems, however, T in fact comprises two components:

\[ T = [\text{<scope}>.] \text{<object identity>} \]

<scope> is a collection of objects that can potentially receive the message. It is optional and, if omitted, the implicit scope is the entire database. <object identity> specifies an object in the context of the specified <scope>. T, therefore, is a qualification expression in the dot (‘.’) notation that should be familiar to programmers. A qualification expression, in fact, is the general form for specifying objects, and thus applies to the parameters of a message as well. More specifically, we define qualification expressions as:

\[
\begin{align*}
\text{<qual-expr>} & \ ::= \text{<object identity>} \\
\text{<qual-expr>} & \ ::= \text{<scope>.<object identity>} \\
\text{<qual-expr>} & \ ::= \text{<query>} \\
\text{<query>} & \ ::= \text{<scope>.<predicate>} \\
\text{<scope>} & \ ::= \text{<qual-expr>}
\end{align*}
\]

<object identity> specify objects in ways we will elaborate below. <query> will be elaborated in the next section.

Recall that object creation introduces objects and assigns them unique identities. These identities are generally not directly available, however, to the user. Users must write, instead, expressions that denote object identities. An <object identity> expression can be:

- a unique global name
- local variable name
- a class name
- a message

**Unique Global Name**

The identity assigned to an object at the time of creation is an internally generated system identity. Many systems, however, allow users to specify (probably as part of the ‘NEW’ message) a unique global name/identifier to be associated with the object. User written expressions can then directly use such names to refer to the objects. We have in fact been doing this in our examples, ie. assuming a global name for objects and using them in example messages (such as ‘B36’ in Figure 6-2).

**Local Variable Name**

The use of variable names to denote objects have in fact been illustrated in numerous preceding examples, particularly involving method body definitions. By ‘local variables’, we mean the instance variables and the formal parameter names in message templates of a given class. Both types of variables hold at run-time particular object identity values. Their appearance as <object identity> may therefore be interpreted as evaluating to the object identity they hold.
Figure 6-3 Local variable names as object expressions

Figure 6-3 shows a class with instance variables in its memory and a formal parameter variable in one of its methods. Note the target specification “T.Owner” in the method body, which uses the formal parameter variable T as <scope> and instance variable ‘Owner’ as <object identity>. The object it denotes is determined as follows. First, ‘T’ is evaluated to the object identity value it holds. Next, in the scope of this object, the instance variable ‘Owner’ is resolved. This simply means retrieving the value of ‘Owner’ in that object’s memory, which then is the desired object.

Class Name

A data class name of course conceptually denotes all possible objects that fits its description. When used as an <object identity>, however, it denotes the collection of existing database objects that are its members (ie. only those objects that have been stored in the database). Thus given the class ‘Bank Account’ as in Figure 6-3, we can write:

Get( 'Bank Account' )

Note that as the scope is omitted, it defaults to the entire database. This message therefore serves to select from among all database objects only those that are members of ‘Bank Account’ and returns them as a collection object.

Note that we say members rather than instances, ie. instances of subclasses, if any, are included. This means that abstract data class names can be used as <object identity> as well.

Message

We have already explained earlier how messages, because they evaluate to objects, may be used to denote objects. Thus they can be used as <object identity>.

6.3 Query Expressions

The previous section tells us how objects may be addressed, and to a certain extent we can achieve object selection through the use of class names and method invocations.
More powerful selection facilities, however, must allow selection based on object contents. For example, we may wish to select only those ‘Bank Account’ objects whose owner is “Smith”. This suggests that we must provide comparison operators such as ‘=’, ‘>’, ‘<’, etc., to allow us to write predicates over object states. We will then need a way to write:

<the owner of this account> = “Smith”

Given a ‘Bank Account’ object, say B, we cannot of course write expressions like

B.Owner.Name

since it presumes knowledge of the internal structure of B (and for that matter the internal structure of the class ‘Person’ as well). Our only recourse is send messages to the object to access its state. Thus, if we assume that the ‘Get-Owner’ method returns a string object representing the owner’s name, we can write:

Get-Owner( B ) = “Smith”

This predication applies to one object and denotes a truth value. This is the form that <predicate> takes. Now we need to apply predicates such as these to a collection of objects (the scope) to cause the selection of only those objects satisfying the predicate. The scope and the predicate together forms a query “<scope>.<predicate>”.

The predicate will typically involve messages targeted at objects in scope. Since the target objects are implicit in the scope, messages in the predicate omit specifying them. Thus, selecting bank accounts owned by “Smith” would be written as in the following illustration (note that the ‘Get-Owner’ message does not have to specify a target):

Get( ‘Bank Account’ ). Get-Owner = “Smith”

The scope: all ‘Bank Account’

The predicate

A message targeted to each object in scope

The query, denoting all objects in scope

Note that a <query> also denotes an object, specifically a collection object, and may therefore be the target of a message.

For example:

Print( Get( ‘Bank Account’ ). Get-Owner = “Smith” )

As further examples of query construction, assume a database populated with objects of classes in Figure 6-4. Assume further that for each class, there is a method ‘Get-X’ to retrieve the value of instance variable X.
**Query**: Get customers who bought a CPU.

There are several ways this query can be constructed. In one, we observe that the required customer objects can be retrieved from transaction objects of CPU sales. A transaction object \( T \) is for a CPU if the following is true:

\[
\text{Get-Pname( Get-Product (T) ) = "CPU"}
\]

The collection of CPU transactions, therefore, is represented by the query:

\[
\text{Get( Transaction ). Get-Pname( Get-Product ) = "CPU"}
\]

We can now use this as the scope to get the desired customers:

\[
( \text{Get( Transaction ). Get-Pname( Get-Product ) = "CPU"} ). \text{Get-Customer}
\]

Alternatively, we observe that product objects are associated with a collection of transaction objects. Thus, we can retrieve all the transactions for CPU from the product object for CPU. The latter is simply the query:

\[
\text{Get( Product ). Get-Pname = "CPU"}
\]

All the relevant transaction(s), therefore, is given by:

\[
( \text{Get( Product ). Get-Pname = "CPU"} ). \text{Get-Transactions}
\]

and finally, the desired customer(s) is expressed by:

\[
( ( \text{Get( Product ). Get-Pname = "CPU"} ). \text{Get-Transactions} ). \text{Get-Customer}
\]

---

**Query**: Get products purchased by customers from Graz.

Again there are several possible constructions, of which we will develop one. The reader may attempt other constructions as an exercise.

We observe that from a customer object, we can retrieve all his/her transaction objects, and from each transaction we can retrieve the product purchased. Of course, we are only interested in customers from Graz, thus we use the query:

\[
\text{Get( Customer ). Get-Ccity = "Graz"}
\]
Object-Oriented Data Model

This then becomes the scope to retrieve transactions:

\[( \text{Get( Customer ). Get-Ccity = "Graz" ). Get-Transactions} \]

This, in turn, becomes the scope to retrieve the desired products:


The notation used here is not that of any particular object-oriented query language, nor is it proffered as one. We use it here only to facilitate description of the principal concepts of object-oriented queries. Real object-oriented systems frequently use more concise notations. For example, the qualification expression above might be written as:

\[ \text{Customer.Get-Ccity = "Graz".Get-Transaction.Get-Product} \]

Qualification expressions are assumed to be left-associative, so parenthesis may be omitted. ‘Get’ may be assumed to be performed on object identities whose objects have not yet been retrieved from the database.

Finally, we note that a collection of objects is also necessarily a set (no two objects can have the same identity in the same collection). Thus, conventional set operations can be applied to them and, in particular, to qualification expressions that evaluate to collections. This allows us to handle queries such as “Get products purchased by customers from Graz or Vienna” simply as a union of products purchased by customers from Graz and products purchased by customers from Vienna:

\[( \text{Customer.Get-Ccity = "Graz".Get-Transaction.Get-Product} \]

\[ \text{UNION} \]

\[ \text{Customer.Get-Ccity = "Vienna".Get-Transaction.Get-Product} \]

Similarly, queries involving conjunctions can make use of set intersection, negation can make use of set difference, and so on.

6.4 Summary

Ad-hoc query construction is not generally considered a powerful feature of object-oriented databases. The reader can see from the foregoing that queries can be cumbersome to construct and requires considerable understanding of object-oriented concepts.

More often, therefore, queries are anticipated by database developers, built into data classes as methods and provided through easy-to-use interfaces for database users. For example, the ‘Customer’ class may have the following method pre-defined:

\[ \text{Get-Products ( T:Customer ) R: Collection ( Set (R, Get( T.Transactions.Get-Product )) )} \]
A user then need only identify a particular customer to see all the products that he/she had purchased. Other similar retrieval methods may be thus embedded into data classes to hide the complexities of database retrieval from the end-user.

Because ad-hoc query facilities are poor, greater onus is on the object-oriented database developer to anticipate uses of the database and to predefine them in user interfaces than if database systems with more friendly end-user query facilities were used.
7 Metalevel Facilities and Database Architecture

7.1 Introduction

In section 14.3 we introduced dynamic binding - the run-time binding of execution code to a message. Static binding is not possible because the execution code for a message selector depends on the target object and the latter is only known at run-time. ‘Print’ messages are good examples of the need for dynamic binding: many objects provide a public method named ‘Print’ but each object may define them differently; thus the code to execute the message “Print( <target>, … )” can only be determined once the class that <target> is an instance of is known.

This turns out to be an important and vital feature of object-oriented systems. The power and economy of static inheritance would otherwise be impaired without it. It allows, for example, methods to be inherited unchanged from a superclass while changing the domain binding of some instance variable(s)-so long as the new domain can react to at least the same messages that the old domain could. This is illustrated in Figure 7-1. Note that even though the subclass redefines the domain binding of the variable ‘Owner’, the inherited method ‘Print’ remains unchanged since dynamic binding will ensure that the message “Print( T.Owner )” will invoke the right code definition. Abstract data classes, in particular, rely on this property to pass down methods to subclasses.

Figure 7-1 Role of dynamic binding in static inheritance

While methods are dynamically bound to messages, note that instance variables are statically bound to their respective domains when a class is defined. There is nothing wrong with this and many object-oriented systems provide little beyond this. Such systems are thus characterised by a static schema, ie. a set of data classes that do not change at run-time; only objects and their states do. When there is a need for a new data class, say a new type of account, the schema must be modified (off-line by a database administrator) by adding a new data class.
For many applications, this suffices. However, there are arguably situations that can benefit from an ability to create data classes at run-time. This would open up, for example, opportunities to write intelligent object-oriented database applications that modify their own schema to adapt to a changing environment. Thus, for the banking example above, it is conceivable that the application includes facilities for end-users to interactively define attributes for a new type of account and cause a new data class representing such accounts to be generated!

The reader might wonder at this point why we should bother with subclassing at all, if all we want to do is redefine a variable’s domain, such as in Figure 7-1. Why not just make the ‘Object’ data class the domain of ‘Owner’ for example? Then any object will be a valid value and there would not be any need to subclass ‘Bank Account’ just to change the domain of ‘Owner’! This is true of course, but it defeats the purpose of typing and of creating different data classes in the first place. Object-oriented data modelling is intended to structure the universe of objects and this means typing objects through data classes (objects with similar attributes).

7.2 Metavariables and Metaclasses

One approach to the dynamic creation of data classes is to extend the idea of abstraction to data classes themselves. That is, just as a collection of similar ADOs is abstracted or described by a data class, we abstract or describe a collection of similar data classes. Then, we can create a data class from its abstract description just as we can create an ADO from a data class.

In describing things, we distinguish between the description formalism and the things being described. The latter are said to be at the ‘object-level’ (the objects of description) while the descriptive constructions are said to be at the ‘meta-level’. Thus for example, we may use Malay sentences to describe the English language, i.e. Malay is used as a metalanguage for English. Or, as we have done in this text, we use BNF-like constructions as a metalanguage for various object-level language constructions like relational calculus. It is possible also that a formalism is used as its own metaformalism, e.g. using English to talk about English sentences.

Making use of this distinction, we could have called data classes “metaobjects”-objects that exist at the meta-level and which describe ADOs. In like manner, we will call the objects describing data classes “metaclasses” (we could use “meta-metaobjects”, but this gets a bit awkward!).

The particular form of metaclass we will discuss here looks very much like a data class except that the domain of an instance variable can be an identifier other than an existing data class name and interpreted as a variable that can take data class names as its value. Note that such an identifier is a variable only at the meta-level and, to avoid confusion with object-level variables, we will refer to it as a “metavariable”. Given a metaclass with metavariables, the idea then is to derive from it a data class definition by dynamically binding metavariables to data class names. This approach therefore introduces a dynamic binding of instance variables to domains.
The creation of a data class from a metaclass, just like the creation of an ADO from a data class, is an instantiation process. That is, data classes are instances of metaclasses and are created in response to ‘NEW’ messages sent to metaclasses. Thus, just as data classes are viewed as cookie cutters or factories for ADOs, metaclasses are factories for data classes.

Defining a metaclass is very similar to defining a data class:
1. Define a unique name for it
2. Define its private memory structure
3. Define its public methods

This is illustrated in Figure 7-2 (assume that we have decided to make ‘Bank-Account’ a metaclass instead). Note two points of difference compared to a data class definition. First, the use of metavariables—‘Class-Of-Owner’ and ‘Class-Of-Sum’—are metavariables that can take existing data class names as values. Second, such metavariables and metaclass names can also be used in message templates as the type of formal message variables.

The ‘NEW’ message template for a metaclass specifies the binding of metavariables to class names. For example, the ‘NEW’ template for the metaclass ‘Bank-Account’ above would be:

`‘NEW’( Bank-Account, Class-Of-Owner: Class, Class-Of-Sum: Class, … ) …`

where ‘Class’ is a system-defined abstract class of classes. A ‘NEW’ message must therefore provide names of existing data classes as parameters. The result will be a data class definition with all occurrences of metavariables, including those in method definitions, replaced by corresponding class names in the message. For example, to create the ‘Student-Account’ data class, issue the message:

`‘NEW’( Bank-Account, Student, Numeric, … )`

The result is illustrated in Figure 7-3. Note that occurrences of metanames are replaced by data class names.
In defining metaclasses, we can use static inheritance. A sub-metaclass will inherit the super-metaclass’ memory structure and methods, unless overridden, and can add new instance variables and methods. This is illustrated in Figure 7-4.

Thus, an object-oriented system supporting static inheritance and metaclasses have two parallel inheritance hierarchies: a metaclass hierarchy and a data class hierarchy, as illustrated in Figure 7-5. All metaclasses inherit from the super-metaclass ‘Class’. Instances of metaclasses, i.e., data classes, are therefore members of ‘Class’ (thus the use of ‘Class’ as the type of metavariables in ‘NEW’ message templates). And as we have seen earlier, all data classes inherit from the superclass ‘Object’ and their instances, i.e. the ADOs, are thus members of ‘Object’.
In such systems, users have a choice of either creating a data class explicitly using data class expressions or creating a data class by sending a ‘NEW’ message to a metaclass. Generally, however, explicit definition of data classes are sufficient for most practical purposes. Metaclasses tend to be used in specialised applications where meta-level abstractions are required to manipulate classes directly, rather than just their objects. The metaclass as described is, in any case, not widely supported amongst existing object-oriented systems. Whether or not they will gain wide support still remains to be seen.

7.3 Architecture of Object-Oriented Database Systems

Let us review now the principal concepts and components constituting an object-oriented database system.

First is to note the very important role played by abstract data classes and static inheritance. An object-oriented DBMS provides a number of system data classes and many of these are defined as abstract classes. The idea of persistent database objects and the general properties governing their creation, manipulation and deletion, is in fact captured as an abstract class called ‘Database_Object’. All database objects are members of this abstract class.

Users can also define abstract data classes to organise their description of an application domain. Through the power and economy of static inheritance, hierarchies of data classes leading to application-specific data classes may be defined. This is illustrated in Figure 7-6.
The role of Abstract Data Classes in Object-Oriented Databases

The architecture of an object-oriented database system is depicted in Figure 7-7.

The database schema defines data classes and their associations with one another. Associations may be expressed through variable-domain bindings and through inheritance (both static and dynamic). The database is the set of persistent data objects that are instances of data classes in the schema. Thus, the schema determines all possible objects that can populate the database.

Users interact with the system by sending messages to data objects and receiving their responses. The types and structure of messages and responses are predefined in the database schema, ie. the public interfaces of data classes (and their instances). The set of public interfaces therefore constitute the Data Manipulation Language (DML) of the system.

The message server is the component of the database system that handles external and internal messages. External messages are those that come from or go to the user, such as a query and its response. Internal messages are those generated and passed between database objects, such as those generated in executing a method body. The message server in both cases is responsible for the proper handling of messages. The database
schema is heavily used in deciding what to do with messages, as it contains the definition of methods and instance variables. The former is needed to execute an incoming message and the latter is needed to get identities of objects to pass messages to.

7.4 Summary and Conclusion

The following features of an Object-Oriented Database System distinguish it from Object-Oriented Programming Languages:

1. Data Objects are persistent, exist independently of user sessions and can be shared between users (Figure 18-8 (a)).

2. Data Objects have states that can change over time and may thus react differently to the same message at different times. Figure 18-8 (b) shows an object’s response to the same message, returning ‘X’ before and ‘Y’ after a state changing transaction has been applied to it.

Coupled with dynamic binding, these features have important consequences in respect of error detection and database recovery (from interrupted or failed transactions). In databases with complex inheritance hierarchies, and particularly when those hierarchies evolve over time in response to changing or new user requirements, it is extremely difficult to fully ‘debug’ database behaviour. The most common fault arises from secondary messages.

For example, suppose we had the following data class:

```hs
class Bank-Account;

instance variables
    Sum-Available: Numeric;
```

![Figure 7-8 Object Persistency](image-url)
Object-Oriented Data Model

methods

Transfer( To: Bank-Account, Sum: Numeric, From: Bank-Account ) R: Logical;
[ + ( To.Sum-Available, Sum );
  – ( From.Sum-Available, Sum );
  Set( R, true )
]

end Bank-Account;

A primary message such as “Transfer( A1, 50, A5 )” generates secondary messages as a result of executing the body - specifically, the “+” message directed to the object “A1.Sum-Available” and the “–” message directed to the object “A5.Sum-Available”. The problem is that the actual objects receiving them are only known at run-time and there is no guarantee that they can handle the messages. An object-oriented run-time system, in general, must therefore be able to detect cases where an object cannot handle a message directed to it and deal with the error appropriately.

For general programming systems, it may be sufficient to just flag the error and terminate execution or return an error code for the program to act on. For a database system, however, this is not enough. In the above example, it may have been that the “+” message was successfully executed but the “–” message failed. Just flagging the error will clearly leave the database in a logically inconsistent state! Thus, the run-time system of an object-oriented database should also have mechanisms to recover from failure and reinstate objects to their states before a failed transaction. The exact nature of such mechanisms is not within the scope of this book, but suffice it to say here that they would be similar in principle to those described for relational databases.
Object-Oriented Data Model