CHAPTER 1

CORBA Fundamentals and Programming - Tutorial Presentation

A Presentation prepared by

Jon Siegel, Director of Domain Technology, Object Management Group

Example Rationale

The products which implement CORBA cover a lot of ground: programming languages from C and C++ to COBOL and Ada; platforms from mainframes to micros to desktops; networks from fiber to satellite links. The tutorial programming example in this presentation showcases both the diversity and the commonality of the ORBs and CORBA development environments on the market today. Today, we'll work it in C and C++; in the book it is worked in eight ORBs, from seven vendors, in the three programming languages C, C++, and Smalltalk. More ORB vendors have worked the example as well; OMG maintains a web page with pointers to every instance we know about. The programmers and authors who wrote the example worked hard to maximize commonality: the Analysis & Design and the IDL file are common to *every* implementation; within each language, almost all of the code is as well. Common code is presented only once; ORB-specific sections detail product-specific portions and go on to explain how to use each programming environment to generate your executable objects. By the end of the day, we'll have covered a lot of information on CORBA programming; there's even more in the book

that we won't have time for. We'll try for the best possible compromise between clarity and coverage in the presentation.

1.1 A Unique Opportunity

We never counted up the total number of platforms covered by the ORBs represented in the book. Two of the ORBs run on more than twenty platforms *each;* due to overlap and the continuous introduction of new platforms it's not practical to put a single number on this!

Platform diversity, even within relatively small companies, is a fact of life -frequently, a specialized platform can deliver low-priced high performance in an area where another platform, albeit more popular, can not. Intentionally designed to be platform-independent, CORBA provides interoperability and portability across platform boundaries.

1.2 POS Example Structure

A chain of grocery stores is changing its information handling. It is installing intelligent cash registers, called Point-of-Sale (POS) terminals, with bar-code readers and receipt printers. These POS terminals are all connected to a single store computer which contains information common to all POS's in the store but potentially different than information in other stores (such as markup, tax policy, and store totals). The store computer answers requests and stores summary data for the POS terminals. All store computers for stores in the chain are, in turn, connected to the central office's depot, which supplies item information to the store such as cost before markup and descriptive information to be printed on receipts. The depot also keeps track of each store's inventory and will, eventually, schedule deliveries to the store based on the inventory.

Here is a "management level" problem statement, stating the rough information flow as a series of tasks.

A. A chain food store has several POS stations connected to a store computer. A cashier turns on the POS station and logs into the store computer stating that the POS system is ready for busines. The store computer runs continuously and is connected to a continuously running central depot computer. It is the job of the

depot to keep track of all inventory and to respond to requests with the chain's cost for each item as well as the taxable status for an item (food, clothes, other). All taxation is calculated at the local store.

B. Each POS station is connected to a barcode scanner and keypad. The barcode scanner outputs a number to the POS station. The keypad can transmit one of five choices to the POS station:

1. Login a new cashier.
2. Print a slip showing the total sales for this POS station since the last login.
3. Print a slip showing the total sales for the entire store since the store computer started
4. Indicate a quantity that applies to the next scanned item. The default quantity is one.
5. Total a sale (a series of one or more grocery items).

F. The store has multiple items, each of which has a barcode. A customer brings a basket of items to a cashier at one of the POS stations. The cashier, who runs each item over a barcode scanner, may enter a quantity greater than one via the keyboard for each barcoded item when more than one of the same item is being purchased. The POS station then prints the sum of all sales to finish the sales slip. The cashier rips off the printed sales slip and gives it to the customer. The cashier bags the order, collects the money, and tells the customer to have a happy day, whether the customer is smiling or not.

G. In order to print each line of the sales slip and to tell the central depot about its inventory, each POS station sends barcode number and quantity to the store computer which passes them on to with the central depot, getting back at least the chain's cost for the item, the tax type for the item, and the item-name. The store computer then adds on a store-specific constant percentage markup for an individual item, which is then reported back to the POS station and printed as the customer's per-item price for the sales-slip. The store computer also returns the taxable price for each item to the POS so that it can figure out the tax at the end. The POS station calculates the amount of the (possibly) multiple items for a single line of the sales slip. A single line printed on the sales slip has the barcode, the item-name, the sales price for each item, the quantity of each item and the total sales price for the line.

H. The store computer keeps track of the total amount of sales during the day. From any POS station, the store manager can ask for a daily report. The POS station reports the total sales since the store computer started. (There is no special console for the store computer; any POS can get the store totals.)

I. The POS station keeps a running tally of all its sales since its login time. The cashier can ask for this running total at any time except during a sale.

J. The taxes are calculated at the store level to allow for varying jurisdictions' policies. The chain management provides a standard tax object that calculates a flat tax for everything that is taxable. In the example, the total taxable amount is calculated by adding up the individual taxable amounts per item. The taxable amount per item is either zero or the full value of the item. The taxes are always calculated on a final taxable sales total, not on an item-by-item basis, to avoid cumulative rounding errors.

K. For the purpose of this example, an item is either taxable or not taxable and the sum of all taxable items is multiplied by a constant percentage to arrive at the tax. This is very simplistic but will do for the example.

What's left out? A lot! Security, for one thing (although the login operation is provided specifically to show where security would fit), and re-ordering an item when warehouse stock runs low. Don't try to use this version of the example to run a real store because it won't work. We concentrated on interfaces, to teach a lot about CORBA in the space we had, and put only minimal functionality into the body of each routine.

1.3 Objects in the Example

One thing we needed to do was divide up our problem space into a number of objects. In keeping with the principles of object orientation, we wanted them to correspond as much as possible to real-life objects; this also helps us divide the functionality in a natural way. The major objects were easy to determine, but it took our programmers and designers a few iterations before the objects supporting the POS were fully stable. The objects in the final design are:

- InputMedia Object
- OutputMedia Object
- POSTerminal Object
- Store Object
- StoreAccess Object
- Tax Object
- Depot Object
- PseudoNameService Object

The slide shows how these objects connect together to make everything work. Every object accesses the PNS. In the slide, we only show one store connected to the depot; the example could easily be extended to more but we haven't taken the time or space to do this. It is possible to run more than one POS using the example code presented here.

In the book, the Analysis and Design presents details for every object: a short glossary definition, a list of state variables (some identified as CORBA attributes), a list of operations (also called methods), and implementation details for each operation. In this presentation, we'll concentrate on one module - AStore - which contains the Store, StoreAccess, and Tax objects. We've selected this module, which contains the largest assortment of different and interesting OMG IDL constructs, to give you as much content as we could squeeze into a one-day presentation. There just wasn't time to cover it all!

1.4 Store object

Glossary

The Store handles multiple POSes, logging them in, keeping track of their sales, and reporting totals on the store and the POSes.

State variables

Totals:	(CORBA Attribute, read only) a structure consisting of two numbers collected since the store computer last started:
StoreTotal	running sum of total sales (including taxes) for all POSes
StoreTaxTotal	running sum of total taxes for all POSes
POSlist	a sequence of structures about POSTerminal objects that have logged into the store. Each structure contains the id of the POSTerminal, the total sales reported by that POSTerminal since last login, and the total taxes collected by that POSTerminal since last login.

Operations

Initialize

Parameters: None; no return

Description: Start up the store when the store computer starts.

Implementation:

1. Set StoreTotal and StoreTaxTotal to zero.
2. Set POSlist sequence to indicate no logged in POSes.

Called from: Start-up

Login

Parameters: input "Id" (i.e. POS id); return reference to StoreAccess object

Description: Register a new cashier/POS with the store and start keeping track of sales and tax totals for that POSTerminal. [Note: this is also where security would go if there were any.] Create a StoreAccess object and return its reference.

Implementation:

1. If there is no POSTerminal with "Id" already in the sequence,
   - Create a StoreAccess object.
   - Add another POSInfo structure to the POSlist sequence, setting its POSId, zeroing the totals, and setting its StoreAccess reference.
2. If there is already a POSTerminal in the sequence with POSId the same as "Id", just zero out the totals.
3. Return the StoreAccess object reference.

Called from: POSTerminal operation "Login"

GetPOSTotals

Parameters: output "POSData"; no return

Description: Return, in the output parameter, the state of the store's current knowledge of the registered POSTerminals (POSlist).

Implementation:

1. Set POSData values to current values in POSList.

Called from: POSTerminal operations "PrintPOSSalesSummary", “PrintStoreSalesSummary”

Comment: A better, encapsulation-design would be to return just a list of POS totals rather that the POSList which contains a lot of other information. We didn’t do that because we wanted to keep the number of methods down for this example.

UpdateStoreTotals

Parameters: input “Id” (i.e., id of POS caller), "Price", "Taxes"; no return.

Definition: Accumulate the parameters into the store state and into the POS state kept by the store.

Implementation:

1. Search POSList sequence for structure containing a POS id equal to parameter "Id".
   - If not found, there is a major inconsistency in the operation of the entire system. [Notify MIS manager by beeper -- not implemented in demo.] Die.
   - If found, add "Price" and "Taxes" to corresponding fields in the structure found.
2. Add "Price" to StoreTotal and add "Taxes" to StoreTaxTotal.

Called from: POSTerminal operation "EndSale".

_get__Totals

Parameters: None; returns the structure “Totals”

Called from: POSTerminal operation "PrintStoreSalesSummary"

Implementation:

1. Note: this is automatically generated for the Store object by IDL compilers since it refer to an IDL attribute. It is declared read-only so only the store can set it.

1.5 StoreAccess Object

Glossary

StoreAccess Object is the intermediary between the POS and the central depot concerning access to the grocery item data base. One such object is created for each POS that logs into the Store.

State variables

depotRef:	reference to chain's depot object.
taxRef:	reference to local tax calculation object.
storeMarkup:	Percentage markup [Note: this is constant for all items in the store. This is quite unrealistic, but to do otherwise would require a local database for all items with markup for each, or at least a designator from the depot as to loss-leader, small, medium, high, or something like that. Definitely not needed for this CORBA demo.]
store_id	Store identification number, used to identify self to central office depot

Operations

Initialize

Parameters: None; no return

Description: Start up StoreAccess object.

Implementation:

1. Set depotRef, taxRef, and storeMarkup.

Called from: Store operation "Login"

FindPrice

Parameters: input "Item" (a barcode), and "Quantity”;
output "ItemPrice", "ItemTaxPrice", and “IInfo”; no return value;

Exception: BarcodeNotFound

Description: Given input parameters, tell depot about inventory reduction and ask it for IInfo (the information about the barcode in the database). Calculate the output parameters from IInfo's cost, markup, and ItemType (the tax type), the latter using the taxRef object.

Implementation:

1. Call "FindItemInfo" operation of depotRef object with input arguments local variable "store_id", passing through received arguments "Item", and "Quantity"; output argument is passed through argument "IInfo" which returns (among other things) the item's cost and item's type.
2. If 1 yields BarcodeNotFound exception, raise exception BarcodeNotFound and quit. Else (i.e., barcode was found), continue.
3. Calculate "ItemPrice" as item's cost times storeMarkup.
4. Call "FindTaxablePrice" operation on taxRef object with input arguments "ItemPrice" and the item's type, receiving back output argument "ItemTaxPrice".

Called from: POSTerminal operation "SendBarcode"

1.6 Tax object

Glossary

Performs tax services such as determining whether an item is taxable and calculating taxes. For the purposes of the demo, it implements a flat tax method, where clothes and food are not taxable but everything else is taxed via a straight percentage of sales price.

State variables

Rate: Percentage charged

Operations

Initialize

Parameters: None; no return

Description: Start-up when store system starts.

Implementation:

1. Set Rate.

Called from: Start-up.

CalculateTax

Parameters: input "TaxableAmount"; returns tax on that amount.

Description: Calculate tax on parameter amount and return as result.

Implementation:

1. Multiply "TaxableAmount" by Rate, round it, return it.

Called from: POSTerminal operation “EndSale”

FindTaxablePrice

Parameters: input “ItemPrice", "ItemType"; return taxable price.

Description: Determine how much, if any, of "ItemPrice" is taxable in this jurisdiction. In this case, food and clothes are not taxable, but everything else is 100% taxable.

Implementation:

1. If "ItemType" is “food” or “clothes”, return 0.
2. Else return "ItemPrice".

Called from: StoreAccess operation “FindPrice”

1.7 Coding the OMG IDL

The A&D gives us a lot of guidance in writing the IDL. We did this on purpose; we knew during the design phase what the next step would be. Because the A&D specifically states object, attribute, and operation names, the main things we need to do here are to translate the A&D constructs to IDL, and define the modularity of the components.

Here are the assumptions that were “built-in” to the A&D as we wrote it, to enable easy transformation into IDL:

1. Each object defined in the A&D will be defined as an IDL interface.
2. State variables are assumed to be implementation details and will not be part of the IDL unless the A&D states they are attributes.
3. To help illustrate IDL syntax, we will use a variety of IDL constructs to specify our objects.
4. We will use capital letters to separate words in identifiers rather than underscores. (Underscores, although legal IDL characters, can cause compilation and portability problems with both C and Smalltalk.)

First, let us consider the best way to modularize the IDL which represents these objects. We define each object as an IDL interface and group logically related interfaces within IDL modules. In creating these groupings we have made some assumptions on how the system will be deployed. We know from the A&D that object instances will exist on separate computers:

> Each POS station is a separate computer in a LAN within the store. It contains the POS object, the InputMedia and the OutputMedia objects.
>
> The Store object, the StoreAccess objects it creates, and the Tax object reside on a separate computer in the same LAN, but on a different computer from the POS stations.
>
> The depot is on a separate computer reachable by WAN.

This deployment strategy suggests that we create three modules, one for each computer system. The analysis suggests these three:

```
module POS
{
    interface POSTerminal;
    interface InputMedia;
    interface OutputMedia;
};

module AStore
{
    interface Store;
    interface StoreAccess;
    interface Tax;
};

module CentralOffice
{
    interface Depot;
};
```

The last object to define in IDL is the Pseudo Name Service. It will be accessed by all the above modules, so it is declared spearately as a stand-alone interface not enclosed within a module.

1.8 OMG IDL for the Store Object

The Store handles messages to Initialize, login a POS, and to report totals, but the A&D operation “Initialize” will not be represented in IDL. The A&D further specifies that the Store will track running totals from all POS Terminals and will contain an attribute that contains these totals.

We used a single attribute struct, instead of two independent attributes, to avoid possible inconsistencies. This is an example of a problem which comes up frequently in the design of distributed systems, where multiple clients can access a single server for both update and retrieval, so we’ll point it out here. Consider the following scenario:

1. POS 1 calls the ‘get’ operation for the store total
2. POS 2 calls UpdateStoreTotals which changes both the store total and the store tax total

3. POS 1 calls the ‘get’ operation for the store tax total

Even if POS 1 calls the get operation for the store tax total immediately after the get for the store total, there can be no guarantee that the totals are consistent because POS 2 is an independent object and can submit an update at any time. Granted, the probability is slight under light load conditions as we have in this example, but accounting systems are not supposed to report inconsistent results under any conditions.

To avoid this problem, we define a struct with two fields, the store total and store tax total, and a single IDL attribute of the struct type: .

```
struct StoreTotals {
    float    StoreTotal;
    float    StoreTaxTotal;
};
readonly attribute StoreTotals Totals;
```

By combining the two values into a single struct which is retrieved via a single operation, the A&D guarantees that the result will be consistent *for single-threaded servers* (which cannot execute multiple requests simultaneously). Extending this guarantee to multi-threaded servers requires thread control via locks and semaphores, which we will neither define nor explain here. They are extremely useful tools for distributed applications, but we’re afraid they just won’t fit into this presentation (or the book, for that matter).

POSTerminals invoke Store’s operation “Login” to start a session with the store. Login assigns a StoreAccess object to the POS and returns a reference to the StoreAccess object. This can be expressed in IDL as:

```
StoreAccess Login(in POS::POSId Id);
```

Note that Login’s return type is the reference to the StoreAccess object. CORBA semantics dictate that Login return an object reference, not a copy of the actual object implementation. We will see later in the day how this is accomplished in C, and C++.

To complete the Store IDL, we create definitions for the StoreId attribute and the operation GetPOSTotals which reports store totals. From the A&D, GetPOSTotals returns “the state of the store’s current knowledge of all of the registered POS stations”.

To support this requirement we define a structure that represents the state of a POS object and a sequence type of these structures. We do not know, nor do we wish to hardcode, the maximum number of POS objects which can use a store object, so an IDL "unbounded" sequence type (an unbounded sequence can dynamically change size at runtime) is used.

```
struct POSInfo {
    POS::POSId      Id;
    StoreAccess     StoreAccessReference;
    float           TotalSales;
    float           TotalTaxes;
};

typedef sequence <POSInfo> POSList;
```

The POSInfo struct was defined to contain all information that the Store maintains about each POSTerminal including a reference to the POS's StoreAccess object. POSInfo is defined in the AStore module aswe'll see shortly.

The last step is to add UpdateStoreTotals, which completes the IDL for the Store:

```
interface Store {
    struct StoreTotals {
        float    StoreTotal;
        float    StoreTaxTotal;
    };

    readonly attribute AStoreId StoreId;

    readonly attribute StoreTotals Totals;

    StoreAccess Login(in POS::POSId Id);
    void     GetPOSTotals(out POSList POSData);
    void     UpdateStoreTotals(
                in POS::POSId Id,
                in float     Price,
                in float     Taxes);
};
```

1.8.1: StoreAcess Object

The StoreAccess object is, from the A&D, “the intermediary between the POS and the central depot concerning access to the grocery item data base. One such object is created for each POS that logs into the Store.” The A&D indicates that StoreAccess has two operations and several state variables. None of the state variables were specified as attributes. Of the operations, only FindPrice needs to be part of the IDL interface, because it is the only operation invoked by remote clients.

The A&D states that FindPrice accepts inputs to specify the item and outputs the price, taxable price, and an “item info”. It also must raise an exception, BarcodeNotFound, if it cannot locate the item in the data base.

We mentioned IDL exceptions briefly a while ago; now (finally!) we’ll use one. We’ll define BarcodeNotFound, in the AStore module, to contain the offending barcode as follows:

```
exception BarcodeNotFound {POS::Barcode item;};
```

FindPrice also has an output parameter of type “ItemInfo”, defined in the A&D as “a structure containing at least ItemInfo.item, ItemInfo.ItemType, ItemInfo.ItemCost, and ItemInfo.Name”. These types are used by several interfaces so we will put the following declaration in the AStore module.

```
enum ItemTypes {food, clothes, other};
struct ItemInfo {
    POS::Barcode  Item;
    ItemTypes     Itemtype;
    float         Itemcost;
    string        Name;
    long          Quantity;
};
```

ItemTypes is expressed as an IDL enumerated list. Its three identifiers represent the set of item types carried by stores in our system (as defined in the A&D, of course). ItemInfo is defined as stated in the A&D, plus an additional member which contains the quantity in inventory for the item (barcode).

FindPrice can now be defined as follows:

```
void     FindPrice(
                in  POS::Barcode Item,
                in  long         Quantity,
                out float        ItemPrice,
                out float        ItemTaxPrice,
                out ItemInfo     IInfo)
raises (BarcodeNotFound);
```

The IDL interface StoreAccess is therefore:

```
interface StoreAccess {

    void     FindPrice(
                in  POS::Barcode Item,
                in  long       Quantity,
                out float      ItemPrice,
                out float      ItemTaxPrice,
                out ItemInfo   IInfo)
    raises (BarcodeNotFound);

};
```

1.8.2: Tax Object

The Tax object performs tax services for the store object. The A&D states that it contains the operations:

Initialize
CalculateTax
FindTaxablePrice

Two state variables are defined in the A&D but, because they are not attributes, they do not appear in the IDL. And as usual, Initialize will not be part of the IDL.

Both IDL operations return floating point numbers that represent the tax and taxable price for an item respectively. The IDL for the Tax object is:

```
interface Tax {

    float CalculateTax(in float  TaxableAmount);

    float FindTaxablePrice(
                in float        ItemPrice,
                in ItemTypes    ItemType);

};
```

1.8.3: Complete Astore module

The completed IDL for the module AStore is:

```
module AStore {

    enum ItemTypes {food, clothes, other};
    typedef long AStoreId;

    struct ItemInfo {
        POS::Barcode     Item;
        ItemTypes        Itemtype;
        float            Itemcost;
        string           Name;
        long             Quantity;
    };

    exception BarcodeNotFound {POS::Barcode item;};

    interface StoreAccess; // forward reference

    struct POSInfo {
        POS::POSId    Id;
        StoreAccess   StoreAccessReference;
        float         TotalSales;
        float         TotalTaxes;
    };

    typedef sequence <POSInfo> POSList;

    interface Tax {

        float CalculateTax(in float       TaxableAmount);

        float FindTaxablePrice(
                in float          ItemPrice,
                in ItemTypes      ItemType);

    };

    interface Store {

        struct StoreTotals {
            float    StoreTotal;
            float    StoreTaxTotal;
```

```
        };

        readonly attribute AStoreId StoreId;

        readonly attribute StoreTotals Totals;

        StoreAccess Login(in POS::POSId Id);
        void    GetPOSTotals(out POSList POSData);
        void    UpdateStoreTotals(
                    in POS::POSId    Id,
                    in float         Price,
                    in float         Taxes);
    };

    interface StoreAccess {

        void    FindPrice(
                in  POS::Barcode Item,
                in  long         Quantity,
                out float        ItemPrice,
                out float        ItemTaxPrice,
                out ItemInfo     IInfo)
        raises (BarcodeNotFound);

    };

};
```

CHAPTER 2

Compiling the IDL

2.1 The IDL Files

In preparation for compiling the IDL, we must partition it into one or more source files. IDL source files are typically named with the sufix ".idl", although this may vary between IDL compilers. We'll put the IDL for each module into a single file, since we have only a few dependencies. (If we had an object which was inherited by a number of different modules, we might put its IDL into a file by itself for easier inclusion.)

Here is the skeleton of each file:

File POS.idl

```
#ifndef POS_IDL
#define POS_IDL

module POS {
...
};

#endif
```

File Store.idl

```
#ifndef STORE_IDL
#define STORE_IDL

#include “pos.idl”

module AStore {
...
};

#endif
```

File Central.idl

```
#ifndef CENTRAL_IDL
#define CENTRAL_IDL

#include “pos.idl”
#include “store.idl”

module CentralOffice {
...
};

#endif
```

File PNS.idl

```
#ifndef PNS_IDL
#define PNS_IDL

interface PseudoNameService
{
...
};

#endif
```

The IDL compiler support C++ style preprocessing so we use “cpp” conditional, define, and include directives. Each file is bracketed by the lines

```
#ifndef name
#define name

#endif
```

where “name” is replaced by a, hopefully, unique string derived from the file name. For example, POS_IDL, is used in the file POS.idl. These three lines protect against multiple-declaration errors.

Notice also that Central.idl and Store.idl use the cpp include statement so the IDL compiler will know of constructs from the other modules.

If your files contain the IDL given at the end of each module-section in this chapter, bracketed by the structure we just presented, then you’re ready to compile your stubs and skeletons starting in the next section.

2.2 Compiling the IDL

In this section, we will look at how to translate the IDL definitions into C++, so that they can be used by the client programmers, and implemented by the server programmers. The commands here are representative of the many ORBs which support C++. To see the exact commands for each ORB, you will have to either refer to the book, or the documentation which comes with each ORB. But the presentation in this section will show you all the details representative of this class of ORBs.

Each IDL file must be compiled, both to check the syntax and to map it into C++ so that it can be used by clients and implemented at the server side. The IDL compiler can be run from the command line as follows:

```
idl -B Pos.idl
idl -B Store.idl
idl -B Central.idl
idl -B PNS.idl
```

The IDL compiler produces C++ code to suit the chosen C++ compiler. For example, it will use C++ exceptions were these are supported, or it use a standard workaround where they are not supported.

A typical IDL compiler produces at least three output files for each IDL file. From the file **`Store.idl`** it produces the following files:[1]

TABLE 0-1

file name	contents
`Store.hh`	header file containing the C++ translation of the IDL. This is included (using #include in the normal way) by both the client and the server C++ code.
`StoreC.cc`	Client stub code. This is compiled and linked with the client.
`StoreC.cc`	Client skeleton code. This is compiled and linked with the server.

These files contain automatically generated code and need not be edited by the programmer. They contain all of the code required to make CORBA compliant remote invocations, including all of the necessary marshalling, unmarshalling and dispatching code. They can be compiled using the chosen C++ compiler, for example:

```
CC  -I<include_dir>   -c StoreC.cc
CC  -I<include_dir>   -c StoreS.cc
```

Normal CC switches include the **`-I`** switch to indicate the location of the standard include files (typically the **`include`** directory). The resulting file **`StoreC.o`** must be linked with any client that uses the Store IDL definitions; and the file **`StoreS.o`** must be linked with the Store server. Example Makefiles will be shown later.

1. The code extensions are generated to suit the chosen C++ compiler. In addition, other file name roots can be specified via a switch to the IDL compiler.

The IDL compiler itself takes a set of switches that can be used to control its actions. The most commonly used switch, **-S**, instructs the IDL compiler to produce a starting point for the C++ class that is to be written to implement an interface. The result is a pair of files containing the implementation class's definition and the definition of its member functions. The server programmer need add only member variables (and optionally other functions) and then code the bodies of the member functions.

For example, to code the Store IDL interface a programmer would add member variables and possible a constructor and destructor to the following code (by converntion, interface Store is implemented by class Store_i):

```
class Store_i : public StoreBOAImpl {
public:
     // (automatically generated) declaration
     // of each member function
};
```

The definition of each member function is also automatically generated, with a null body that can be filled in by the programmer (naturally, during early development, a programmer may decide to fill in only a subset of the function bodies).

Another switch (**-R**) is used to register the IDL definitions in the Interface Repository.

CHAPTER 3

Language Mappings, Part 1

3.1 Role of a Language Mapping

So far, we've only seen how to write our CORBA interface definitions in OMG IDL. Since these definitions are programming-language independent, how are we going to access these interfaces from our code?

Programming Language Mappings "map" (that is, define one-to-one correspondences) from OMG IDL constructs to programming language constructs. These constructs tell client and object-implementation writers what to write to invoke an operation on a CORBA object. In a non-object oriented language like C, invocations typically map to function calls; in object-oriented languages like C++ and Smalltalk, mappings try to make CORBA invocations look like language-object invocations.

It would be really nice if we could go the other way as well, and generate the IDL automatically from either the client or object implementation source code. You can, and some products already do it. However, they cannot take advantage of everything that IDL has to offer: one-way invocations; attributes including read-only and read-write; DII invocations including asynchronous, deferred synchronous, and multiple deferred synchronous modes; and the other special IDL features which become important as your distributed system grows in complexity. So, while this feature will save a lot of time for a lot of programmers, the best of the bunch

will still code IDL by hand where its advanced properties let them tune their distributed system and make it sing.

The first section of this chapter summarizes aspects common to all language mappings. The remainder of this chapter covers the C language mapping, followed by C++.

What about other languages? The Smalltalk mapping is described in the book. Language with pointers, structures, and dynamic memory allocation support IDL mappings most easily, but languages without them (FORTRAN, for example) support mappings as well. An Ada mapping has just been specified; contact OMG for a copy of the specification if you're interested. There is current work on a COBOL binding as well.

3.1.1: Language Mapping Functions

A language mapping has a lot to do. First, it has to express all of the constructs of IDL including

- basic and constructed data types, constants, and objects;
- operation invocations, including parameter passing;
- setting and retrieving attribute values; and
- raising exceptions and handling exception conditions.

To ensure interoperability, *every* language mapping must provide a mapping for *every* IDL datatype. If some IDL datatypes were not supported by every language, operation signatures which used them would not interoperate everywhere.

There is also a converse problem, of language datatypes which do not map directly to IDL datatypes. For example, COBOL includes PIC 99 and VSAM record types which do not correspond to any current IDL type. In late 1995, OMG members were evaluating possible extensions to the current roster of IDL types; candidate specifications would provide more natural mappings to datatypes found in COBOL and Ada, as well as various international character sets. But, since interoperability does not require that every language datatype have a corresponding IDL type, we expect that many unusual language types will continue to lack IDL types even after this effort concludes. For up-to-date information on this, contact OMG or your ORB vendor.

There must be a datatype to represent the object reference. This representation is opaque; that is, the datatype is represented by a language-specific datatype but the program does not (and, generally, can not) interpret the value which the type contains. Object references may be passed as parameters or return values in invocations, and may be passed to ORB and BOA operations.

Object invocations are mapped in various ways. In C, the static invocation interface mapping requires the client to insert an object reference into the mapped function call to specify the target. In C++ and Smalltalk, both object-oriented languages, a CORBA invocation looks like a C++ or Smalltalk object invocation.

For the DII, mappings vary. In C, the mapping follows the normal definitions even though the target is a pseudo-object. In C++, attribute operations replace **add_argument**. The CORBA 2.0 specification speculates that a mapping for a dynamic language like LISP could make a dynamic invocation look like an invocation of a language object. Unfortunately, no such mapping exists today (at least that we know about.)

The mapping must also provide access to ORB and BOA functionality which, although expressed in PIDL, is not always accessed as the call which would be constructed from it. In fact some of the PIDL definitions are left incomplete, to be fully defined only in the various language mappings. Similarly, the mappings provide access to the DII including the various deferred invocation modes, and the DSI for the server side.

The last thing the language mapping does is fix responsibility for memory allocation and freeing among the client, the stub, and the ORB. We'll point out how this happens in the various languages as we work our way through the next three chapters.

And finally, remember that C and C++ are both generated by the same compiler. Some ORB vendors required mappings for those two languages which coincided in the various functional areas including memory management. This requirement, in the end, resulted in some changes in an older version of the C mapping being required by the C++ mapping. All of the products represented in this book use the current mappings.

We admit in advance that the simple examples we'll use to illustrate the three mappings are rather dry. We've done this on purpose: Since the second half of the book contains realistic examples worked in detail in all three language mappings, the illustrations in this part have been kept as simple as possible to make the principles

clear. For each language, we'll show an interface and invocation, an inheritance example, an attribute, and an exception. We'll also discuss how memory management is handled for each.

CHAPTER 4

IDL to C++ Mapping

The OMG defines a standard mapping from IDL to C++. That is, it defines a set of rules for how any particular construct in IDL maps to C++, so that client C++ programmers make calls to CORBA objects, and server programmers can implement them in C++. This mapping is carried out automatically by translation programs, so called IDL compilers, that ship with CORBA products.

Much of the mapping is straightforward. IDL interfaces, for example, are translated into C++ classes. Some of the mapping requires C++ programmers to become familiar with types, in particular **`_var`** and **`_ptr`** types, that are generated by the IDL compiler. Once a programmer is familiar with these types, all of the commonly used parts of the mapping are easy to understand.

The full mapping assumes that your C++ compiler is complient with the work of the ANSI/ISO C++ standardization committee. Where this is not the case, the C++ mapping uses older compiler features to compensate (for example, where namespaces are not available, it uses classes).

4.1 C++ Mapping Fundamentals

4.1.1: Mapping for Basic Datatypes

Before we can do anything, we have to know how the defined IDL types (Appendix B) map to C++ types. Because the IDL types are defined in a system-independent way, they cannot map directly to C++ types such as long, short, or float which may be defined differently on different systems. So the C++ mapping, like the C mapping, defines them in terms of CORBA_long, CORBA_short, and so on which are defined in a system-specific header file which ensures that the CORBA requirements are met.

So the header file maps, for example,

- IDL *long* to C++ *CORBA::long;*
- IDL *short* to C++ *CORBA::short;*
- IDL *float* to C++ *CORBA::float;*
- IDL *double* to C++ *CORBA::double.*

and so on. A header file assures that, on any system, CORBA::long always maps to a 32 bit integer; a CORBA::float always maps to an IEEE 32-bit floating point number, and so on through the list.

4.1.2: Mapping for other Types

The IDL provides mappings for string types, structured types (struct, union, and sequence), and arrays. It also discusses use of **Typedefs**, and mapping for the **any** type. Although we will not go into the details here, the tutorial example deliberately includes a **struct**, a **sequence**, and an **enum** to illustrate their mapping.

T_var types: The C++ mapping provides a **_var** type for almost every type which automates memory management. Termed **T_var** types, their names are constructed by adding **_var** to the name of the type. Most helpful for variable-length structured types, **T_var** types are also defined for fixed-length structured types to allow a more consistent programming style. Thus you can code in terms of **T_var**s for structured types uniformly, regardless of whether the underlying types are fixed- or variable-length. T_vars are designed specifically for allocation on the stack; when used this way, all storage used by the type is automatically freed when the variable goes out of scope.

One place the T_var type shows its benefit when you change the length of a variable-length type. This is helpful, for example, when simply copying a longer string into an existing string , but really shows its worth when used for out and inout variables which change size during a CORBA invocation. Here, the automatic memory management provided by the **_var** type allows the client, the ORB and the object implementation to work together to deallocate memory used by the **inout** and **out** parameters at the start of the call, and allocate exactly the memory required by the returning values for all of these types. (Although **out** parameters are not read by the object implementation before being set, they may contain values--and therefore occupy memory--left over from a previous call.) Without the help provided by the **_var** class, you would have to keep track of this memory, and perform at least some of this de-allocation, yourself.

4.1.3: Mapping an Operation

Remember the simple interface we used in the C example? It was

```
interface example1 {
    long op1 (in long arg1);
}
```

In C, this example generated about 5 lines of declarations which told us how to invoke op1 on an example1 object. Recall we had to specify the object reference and the environment explicitly. In C++, the invocation looks object-oriented:

```
// C++
// Declare object reference:
     example1_var   myex1;
     CORBA::long    mylongin, mylongout;

// code to retrieve a reference to an
// example1 object and bind it to myex1  . . .

     mylongout = myex1->op1 ( mylongin );
```

The CORBA object reference in C++ can take one of two forms: pointer (**_ptr**) or variable (**_var**); the **_var** form is memory-managed automatically analogously to the **T_var** structured variable types we just mentioned. If you're building relatively coarse-grained objects, the kind most applications end up distributing around the network, you should use the **_var** form because **_var** object references are automatically destroyed when they go out of scope. But if you're building an application with many fine-grained objects (and using CORBA to ensure inter-language

portability, for example), you may have to use the **_ptr** form to avoid memory management operations which may hamper performance by occurring at inopportune moments. The cost of using **_ptr** object references is that you have to keep track of them, and free their memory yourself to avoid memory leaks.

4.1.4: Passing an Object Reference

Next, an example of passing an object reference. This is the same example we used in the last chapter on the C mapping:

```
#include "example1.idl"
interface example2  {
    example1 op2 ( ) ;
};
```

op2 is an operation with no input parameters which returns the object reference to an **example1** object. This is pretty close to the way your client code will be passed *all* of its object references. In the example, later, we'll write what we call a PseudoNameService which works just like this. The C++ looks like this:

```
// C++
// Declare object references:
     example1_var   myex1;
     example2_var   myex2;

// code to retrieve a reference to an
// example2 object and bind it to myex2  . . .

     myex1 = myex2->op2 ( );

// now we have an example1 object reference
// bound to myex1
```

4.2 Mapping for Attributes

Each read-write attribute maps to a pair of overloaded C++ functions, one to set the attribute's value and the other to get it. The set function takes a parameter with the same type as the attribute, while the get function takes no parameters but returns the same type as the attribute. If your attribute is declared **readonly** in its IDL, the compiler generates only the get function.

So, our attribute example

```
interface example4 {
    attribute float radius
}
```

would be used like this:

```
// C++
// Declare object reference:
    example4_var    myex4;
    CORBA::float    myfloatout;

// code to retrieve a reference to an
// example4 object and bind it to myex4  . . .

// set:
    example4->radius(3.14159);

// get:
    myfloatout = example4->radius();
```

4.3 Mapping and Handling Exceptions

We'll use the IDL from the example later in this book to illustrate basic exception handling. The IDL defines the exception **BarcodeNotFound** in **module AStore**:

```
module AStore {
   . . .
   exception BarcodeNotFound {POS::Barcode item;};
   . . .
}
```

It's actually used in the depot module, in the following C++ code:

```
void Depot_i::FindItemInfo
             ( AStore::AStoreId StoreId,
               const char* Item,
               CORBA::Long Quantity,
              AStore::ItemInfo*& IInfo) {
    IInfo = new AStore::ItemInfo;
    if (m_items.Locate(Item,*IInfo)) {
    }
    else {
      // Raise the exception here
      throw(AStore::BarcodeNotFound(Item));
    }
}
```

This is about as simple as it gets, at least for the programmer. Behind the scenes, the exception maps to a C++ class defined in the **CORBA** module and is a variable-length struct which self-manages its storage. Declarations are in the **.h** file, so you don't have to the declaration of the exception type into your code yourself.

Recall that we have defined both systems exceptions and user exceptions in CORBA. There is a base Exception class, defined in the **CORBA** module. The **SystemException** and **UserException** classes both derive from this, and each specific system exception derives from **SystemException**. This hierarchy allows any exception to be caught by simply catching the **Exception** type:

```
//C++
try  {
. . .
} catch (const Exception &exc) {
. . .
}
```

This approach looks simple so far, but you have to narrow to either **UserException** or **SystemException** yourself. Alternatively, all user exceptions can be caught by catching the **UserException** type, and all system exceptions by catching the **SystemException** type:

```
//C++
try  {
. . .
} catch (const UserException &ue) { . . . }
} catch (const SystemException &se) { . . . }
}
```

4.4 Server Side Mapping

When you write your server, you will be implementing a class which executes the methods you wrote in your IDL file.

However, this class can not have the same name as the interface, because the interface class is an abstract base class which cannot be realized. Implementation classes are created in differern ways by different ORBs; the most common methods are inheritance from the abstract base class, or from some other class related somehow, and delegation.

4.4.1: Using C++ Inheritance for Interface Implementation

Each IDL compiler creates its implementation classes in its own way, optimizing differently for the platforms, C++ compilers, and markets that the company targets. Many--most, perhaps, at least now--derive implementation classes by inheritance, although the base class choice varies widely. (Some use delegation instead of inheritance; other methods are possible too.) We'll go through a basic example here so that you know what to expect when you code your server side, but caution you that details will definitely differ from what you see here.

For our example, we'll look at an implementation class derived from a generated base class based on the OMG IDL interface definition. The generated base classes are known as *skeleton classes*, and the derived classes are known as *implementation classes*. Each operation of the interface has a corresponding virtual member function declared in the skeleton class. The signature of the member function is identical to that of the generated client stub class. The implementation class provides implementations for these member functions. The BOA invokes the methods via calls to the skeleton class's virtual functions.

We'll use this IDL interface for the example in this section:

```
// IDL
interface A
{
    short op1();
    void op2(in long l);
};
```

Suppose our (hypothetical) IDL compiler generates an interface class **A** for this interface. This class contains the C++ definitions for the typedefs, constants, exceptions, attributes, and operations in the OMG IDL interface. It will have a form similar to the following:

```
// C++
class A : public virtual CORBA::Object
{
  public:
     virtual Short op1() = 0;
     virtual void op2(Long l) = 0;
     ...
};
```

Some ORB implementations might not use public virtual inheritance from **`CORBA::Object`**, and might not make the operations pure virtual, but the signatures of the operations will be the same.

On the server side, the IDL compiler will generate a skeleton class. This class is partially opaque to the programmer, though it will contain a member function corresponding to each operation in the interface.

```
// C++
class ABOAImpl : public A
{
  public:
     // ...server-side implementation-specific
     //  detail goes here...
     virtual Short op1() = 0;
     virtual void op2(Long l) = 0;
     ...
};
```

To implement this interface, you must derive from this skeleton class and implement each of the operations in the OMG IDL interface. An implementation class declaration for interface A could take the form:

```
// C++
class A_impl : public ABOAImpl
{
  public:
     Short op1();
     void op2(Long l);
     ...
};
```

4.5 C++ Mapping Summary

That's all we're going to say here about the C++ mapping. We've covered the basics--mapping to types, interfaces, operations, exceptions, and a bit about mapping to the server side. To add the next level of detail would suddenly expand this chapter to five times its size, so we'll leave that step to the vendors who provide manuals for IDL-to-C++ compilers. (You could check out the specification in OMG's CORBA manual if you wanted, but you'd find that it was written for IDL compiler *writers*, and not compiler users!)

Notice how IDL can map naturally to an object-oriented language like C++. One reason is that the operation syntax and format maps naturally; another is that the object-oriented nature of C++ allows the mapping to do a lot of work behind the scenes. The OMG mapping document is a long one, full of detail telling the compiler writer how to generate classes "under the covers" which do the work that lets you work with such short declarations in your own code.

In the next chapter, we'll cover the Smalltalk mapping. Smalltalk, like C++, is an object-oriented language. But unlike either C or C++, Smalltalk is an environment brings with it an entire execution environment.

CHAPTER 5

Coding the Store in C++

5.1 Coding Store, StoreAccess and Tax in C++

Now we need to code the functionality of the store classes. There's almost nothing ORB-specific about this, so we'll do it all in this section. Introduction

The source code for the store has been partitioned into the following files:

FILE NAME	CONTENTS
Store_i.h	class definitions
Store_i.cc	class implementations
Srv_Main.cc	main program control

In our implementation, each store executes a single instance of the Store_i and Tax_i classes. The Store_i object constructs a separate instance of StoreAccess_i for each Point Of Sale (POS) Terminal. For simplicity, all Store_i, Tax_i, and StoreAccess_i object instances execute within a single process.

The following sections describe the logic of the three classes and the 'main' program function.

5.1.1: Tax_i Class

Tax_i is the implementation of the IDL interface ‘Tax’ defined in Store.idl. It is derived from a base class generated by the IDL compiler and contains members and member functions called out in the A&D. To support the A&D requirements we will take the declaration shown in section 28.1 and add a private data member and a constructor.

To implement Tax_i we start with the following observations:

- Tax_i needs a data member which corresponds to the state variable ‘Rate’ in the A&D:

CORBA::Float m_regionRate;

m_regionRate will be a ‘private’ data member of Tax_i.

- The state variable taxTotal called out in the A&D is actually calculated by CalculateTax each time it is invoked. Because there is no need to keep this value across calls, it was implemented as an automatic variable of CalculateTax.
- We know from the A&D that the StoreAccess object will use the tax object to calculate item prices. The requirement for this is derived from the statement in the A&D for the StoreAccess object:

Call “FindTaxablePrice” operation on taxReference object with input arguments ItemPrice and itemTaxType, receiving back output argument item_tax_Price.

To support this requirement, Tax_i will use the PseudoNameService object to ‘name’ itself so clients (e.g. StoreAccess) can ‘connect’ to it.

Based on these observations, the completed declaration for the Tax_i class is:

```
class Tax_i : public TaxBOAImpl
{
private:
     CORBA::Float       m_regionRate;
public:
     Tax_i(PseudoNameService_ptr pns,
        AStore::AStoreId      StoreID);

     virtual CORBA::Float CalculateTax(
        CORBA::Float       TaxableAmount);

     virtual CORBA::Float FindTaxablePrice(
        CORBA::Float       ItemPrice,
        AStore::ItemTypes  Itemtype);
};
```

Constructor

The Tax_i constructor accepts two arguments: a reference to the name service object and the store Id:

```
Tax_i::Tax_i(PseudoNameService_ptr pns,
            AStore::AStoreId      storeID)
```

The constructor uses the PseudoNameService method “BindName” to associate a ‘name’ (i.e. a string) to the instance being constructed. The call to BindName is bracketed by a “try” block. This is how we can detect errors such as CORBA::COM_FAILURE, which would indicate a communications failure.

To allow multiple store programs to run within the same “namespace”, the Tax_i instance’s name is the concatenation of “Tax_” and the store Id. This works because each instance of Tax is used by exactly one store object.

The constructor then sets the tax rate to 5%. To keep the code simple we initialize the tax rate member m_regionRate from the C++ const region_rate defined in store_i.cc.

The complete implementation of the constructor is:

```
Tax_i::Tax_i(PseudoNameService_ptr pns,
             AStore::AStoreId storeID)
{
   // Register the object with the name server
   char regstr[255];
   sprintf(regstr,"Tax_%ld",storeID);
   try {
      pns->BindName(regstr,this);
   }
   catch(...) {
      cerr << "Trouble Binding Tax server" << endl;
   }

   // set tax rate applied to taxable goods
   m_regionRate = region_rate;
}
```

CalculateTax

The virtual member function CalculateTax computes the tax from m_regionRate and the 'in' argument TaxableAmount. It then returns the tax amount.

```
CORBA::Float Tax_i::CalculateTax(
     CORBA::Float          TaxableAmount)
{
   return TaxableAmount* m_regionRate;
}
```

FindTaxablePrice

FindTaxablePrice encapsulates the algorithm for computing the 'taxable price' of an item. In our implementation, the whole price of items of type "other" is taxed; all other item types are not taxed. Therefore, FindTaxablePrice returns either the value of the input argument price for item type 'other' or 0.0 for all other item types:

```
CORBA::Float Tax_i::FindTaxablePrice(
                    CORBA::Float         Price,
                    AStore::ItemType Itemtype)
{
   CORBA::Float taxprice;

   if (Itemtype == AStore::other)
      taxprice = Price;
   else
      taxprice = 0.0;
   return taxprice;
}
```

5.1.2: Store_i Class

The Store_i class is more sophisticated than the tax class in that it services all POS Terminals in the store and keeps track of sales information for the store. POS objects ‘login’ to the store to activate a session with a StoreAccess_i instance. Store_i also contains methods for obtaining sales information.

First, we’ll look at an overview of the implementation class, Store_i:

```
class Store_i : public StoreBOAImpl {
public:
   virtual AStore::AStoreId StoreId();

   virtual AStore::Store::StoreTotals Totals();

   virtual AStore::StoreAccess_ptr Login(
       POS::POSId           Id);

   virtual void GetPOSTotals(
       AStore::POSList   *&POSData);

   virtual void UpdateStoreTotals(
       POS::POSId           Id,
       CORBA::Float         Price,
       CORBA::Float         Taxes);

};
```

To implement Store_i we first add private data members as specified in the A&D:

```
AStore::AStoreId             m_storeID;
CORBA::Float                 m_storeTotal;
CORBA::Float                 m_storeTaxTotal;
CORBA::Float                 m_storeMarkup;
AStore::POSList              m_POSTerminals;
PseudoNameService_var        m_pns;
```

Store_i uses m_pns (which is a reference to the name service object) to ‘name’ itself.

There are several places within Store_i where we will need to locate a particular POS in the m_POSTerminals list. To reduce the amount of redundant code and minimize coding errors we will encapsulate this algorithm in a private function which accepts a POSId as an argument and returns an index into m_POSTerminals.

```
CORBA::ULong locate_POS_entry(CORBA::Long);
```

The methods StoreId, StoreTotal, and StoreTaxTotal simply return the current values of m_storeID, m_storeTotal, and m_storeTaxTotal respectively. They are acces-

sor methods that correspond to the readonly attributes defined in the IDL (in the file Store.idl):

```
readonly attribute AStore     StoreID;
readonly attribute float      StoreTotal;
readonly attribute float      StoreTaxTotal;
```

The following sections provide a detailed look at the implementation of the constructor and remaining member functions.

Constructor

Store_i's constructor implements the processing defined as the operation 'Initialize' in the Store Analysis & Design. The constructor names the instance using the PseudoNameService method BindName. As with the Tax_i constructor, the name is the concatenation of "Store_" and the store Id. Finally, the instance members m_storeTotal and m_storeTaxTotal are set to 0 and m_storeID and m_storeMarkup values are set from the in parameters. The Store_I constructor begins:

```
Store_i(
     PseudoNameService_ptr    pns,
     AStore::AStoreId         storeID,
     CORBA::Float             storeMarkup)
{
     // Register the object with the name server
     char refstring[1024];
     sprintf(refstring,"Store_%ld",storeID);
     m_pns = PseudoNameService::_duplicate(pns);
     try {
          pns->BindName(refstring,this);
     }
     catch(...) {
          cerr << "Trouble Binding " << refstring
               << endl;
     }

     m_storeTotal    = 0;
     m_storeTaxTotal = 0;
     m_storeMarkup   = storeMarkup;
     m_storeID       = storeID;
     // ...
}
```

The last activity of the constructor is to initialize the sequence m_POSTerminals. Note that m_POSTerminals was defined in the IDL to be an unbounded (e.g. dynamic) sequence of POSInfo structs:

```
typedef sequence <POSInfo> POSList;
```

POSList is mapped to a C++ class that provides the ability to get and set the length and iterate over the elements using the '[]' operator (i.e. the operator '[]' has been overloaded to return a C++ reference to the POSInfo object at the specified index location in the sequence). The POSList class has the following form:

```
class  POSList {
public:
     ...
     CORBA::ULong length() const;
     AStore::POSInfo& operator[](CORBA::ULong index)
     ...
private:
     AStore::POSInfo * data;
     ...
};
```

Refer to the C++ Language Mapping chapter, and additionally your ORB vendor's documentation, for a discussion of sequences and their mapping in C++.

The member function length and the '[]' operator allow us to initialize m_POSTerminals using the loop shown below. StoreAccessReference is initialized by calling AStore::Store::_nil(). As mentioned above, _nil() enables different ORBs to represent a 'nil' Store_ptr or nil object reference differently. The remainder of Store_i's constructor is:

```
CORBA::ULong len      = m_POSTerminals.length();
for (CORBA::ULong i = 0; i < len; i++)
{
     // EMPTY is '#define -1'
     m_POSTerminals[i].Id = EMPTY;
     m_POSTerminals[i].StoreAccessReference =
          AStore::Store::_nil();
}
```

Login

Login assigns a StoreAccess_i object to the POS and resets the POS totals to zero. Login first calls the private member function LocatePOSEntry to obtain a 'slot' in

the m_POSTerminals sequence. It then initializes the Id, TotalSales, and TotalTaxes fields:

```
CORBA::ULong loc = LocatePOSEntry(Id);

m_POSTerminals[loc].Id         = Id;
m_POSTerminals[loc].TotalSales = 0;
m_POSTerminals[loc].TotalTaxes = 0;
```

Login next checks to see if a StoreAccess object exists, and if needed, constructs a new instance. It returns a reference to the StoreAccess object by calling the Store-Access class method _duplicate because the return value will be deleted by the ORB as dictated by the C++ language mapping.

```
// check to see of a StoreAccess object exists for
// this m_POSTerminal allocate new one if needed.
if (CORBA::is_nil((AStore::StoreAccess_ptr
)m_POSTerminals[loc].StoreAccessReference))
{
     // create a local instance of the
     // StoreAccess Object
      m_POSTerminals[loc].StoreAccessReference =
          new StoreAccess_i(
                m_pns
               ,this
               ,m_storeMarkup);
      if (CORBA::is_nil((AStore::StoreAccess_ptr )
         m_POSTerminals[loc].StoreAccessReference))
         cerr << "Store_i::Login: Unable to create
StoreAccess object for POS Login" << endl;
}
return AStore::StoreAccess::_duplicate
     (m_POSTerminals[loc].StoreAccessReference);
```

LocatePOSEntry

This private method encapsulates the details of searching the m_POSTerminals sequence for an available entry. It detects when an entry for the specified POS Id

exists and reuses that slot, which prevents creation of a new StoreAccess reference when one already exists.

GetPOSTotals

This method returns the m_POSTerminals sequence to the caller. Note that the argument is an ‘out’ parameter; the semantics of CORBA remote method invocations requires that we allocate and return a copy of m_POSTerminals. Allocating memory is necessary for return values and out arguments that are ‘variable’ length types such as strings, object references, and sequences.

```
void Store_i::GetPOSTotals(
          AStore::POSList *&POSData)
{
   POSData = new AStore::POSList(m_POSTerminals);
}
```

UpdateStoreTotals

UpdateStoreTotals is called to update the running totals for the POS terminal specified by the input parameter Id. It calls locate_POS_entry to locate the entry in m_POSTerminals then updates both the POS totals and the store totals:

```
void Store_i::UpdateStoreTotals(
                        CORBA::Long  Id,
                        CORBA::Float Price,
                        CORBA::Float Taxes)
{
   CORBA::ULong i = locate_POS_entry(Id);
   if (i != EMPTY)
   {
      m_POSTerminals[i].TotalSales += Price;
      m_POSTerminals[i].TotalTaxes += Taxes;
      m_storeTotal                 += Price;
      m_storeTaxTotal              += Taxes;
   }

   else
      cerr << "Store_i::UpdateStoreTotals: Could not
locate POS Terminal " << Id << endl;
}
```

5.1.3: StoreAccess_i Class

StoreAccess provides a mechanism for managing POS sessions and it is the intermediary between the POS and the central depot concerning access to the grocery item data base.

First, we'll look at an overview of the StoreAccess_i class:

```
class StoreAccess_i : public StoreAccessBOAImpl
{
public:
   virtual void FindPrice(
          const char          *Item,
          CORBA::Long          Quantity,
          CORBA::Float&        ItemPrice,
          CORBA::Float&        ItemTaxPrice,
          AStore::ItemInfo          *&IInfo);
};
```

Constructor

The StoreAccess_i constructor sets StoreAccess' state variables from the input parameters. This includes setting m_store. We will call _duplicate to assign a copy of the input argument store. For example:

```
m_store =
     AStore::Store::_duplicate(store);
```

The issues here are similar to those discussed above for GetPOSTotals, store is 'owned' by the ORB and will be de-allocated after the constructor returns. By calling _duplicate we insure that m_store will not 'go out of scope'. The complete implementation or the constructor is:

```
StoreAccess_i::StoreAccess_i(
     PseudoNameService_ptr pns,
     AStore::Store_var     store,
     CORBA::Float          markup)
{
     m_storeMarkup = markup;
     try
     {
          char refstr[255];
          AStore::AStoreId id = pStore->StoreId();
          sprintf(refstr,"Tax_%ld",id);
          m_tax= AStore::Tax::_narrow(pns->
               ResolveName(refstr));
          m_depot=
               CentralOffice::Depot::_narrow(pns->
                    ResolveName("Depot"));
          m_store =
               AStore::Store::_duplicate(store);
     }
     catch(...) {
          cerr << "Trouble finding tax, store, or depot
" << endl;
     }
}
```

FindPrice

FindPrice computes the price and taxable price for the quantity of the item specified by the barcode. It also returns an ItemInfo struct which it retrieves from the central depot. The price (ItemPrice) is computed as the cost times markup percent as required in the A&D:

3. Calculate "ItemPrice" as item's cost times storeMarkup.

The A&D and IDL specify that FindPrice should raise a BarcodeNotFound exception if the input barcode is not known to the depot. Because Depot_i::FindItemInfo itself raises this exception, we simply let the exception propogate to the caller.

```
void StoreAccess_i::FindPrice(
          const char           *Item,
          CORBA::Long           Quantity,
          CORBA::Float&         ItemPrice,
          CORBA::Float&         ItemTaxPrice,
          AStore::ItemInfo         *&IInfo)
{

     if (!CORBA::is_nil((const
     CentralOffice::Depot_ptr)m_depot))
     {
          AStore::ItemInfo *i2;
          m_depot->FindItemInfo(
                    m_store->StoreId(),
                    Item,
                    Quantity,
                    i2);

          IInfo      = new AStore::ItemInfo;
          *IInfo     = *i2;
          ItemPrice
               m_storeMarkup * IInfo->Itemcost;
          ItemTaxPrice = m_tax->FindTaxablePrice(
          ItemPrice,
          ((AStore::ItemInfo *) IInfo)->Itemtype);
     }
}
```

5.1.4: Srv_Main.C

The store program expects the user to supply a store number and markup percent as command line parameters. Main first validates that the correct number of arguments were supplied, and then initializes the ORB and BOA. Main initializes the ORB and BOA by calling the methods CORBA::ORB_init and BOA_init. This is typically the first activity of a CORBA application.

```
if (argc<3) {
      cerr << “usage: “ << argv[0] << “<Store Number>
<Markup>” << endl;
      return 1;
}

CORBA::ORB_ptr orb  = CORBA::ORB_init(argc, argv);
CORBA::BOA_ptr boa  =
     orb->BOA_init(argc, argv, “”);
```

Main then constructs local (i.e. within the current address space) instances of the Tax_i and Store_i classes. Note that this may be accomplished differently by each ORB. After the two objects are constructed, all application related activity is initiated by clients.

The three local CORBA objects tax1, store1, and pns are declared as follows:

```
AStore::Tax_var           tax1;
AStore::Store_var         store1;
PseudoNameService_var     pns;
```

Note that each of these CORBA objects have been declared as ‘_var’ types. The var classes manage the allocation and de-allocation of stub and implementation instances, simplifying the application code. Tax1 and store1 will ‘own’ instances of the implementation class and pns will hold an instance of a stub (returned by FindPNS).

Tax_i and Store_i both take a PseudoNameService reference as an argument. We will call FindPNS to obtain a reference to the name server. Note that this processing is done within a C++ try block (shown below) to allow us to catch any exceptions raised by FindPNS or the constructors. If an exception is raised then control jumps to the C++ catch block.

We verify that FindPNS was successful by ensuring that the return value is non-nil. The Boolean method CORBA::is_nil operates on an object reference and returns TRUE if the reference is ‘nil’ as defined by the ORB. The local implementation instances of Tax_i and Store_i are instantiated by invoking the C++ allocator ‘new’ (this may vary between ORB implementations):

```
try
{
     pns=FindPNS(orb);
     if (CORBA::is_nil(pns))
     {
           cerr << “Unable to get a reference to the
name service” << endl;
          return 1;
     }


     tax1 = new Tax_i(pns,atof(argv[1]));
     store1 = new
        Store_i(pns,atof(argv[1]),atol(argv[2]));
}
catch(...)
{
     cerr << “ERROR Starting Server” << endl;
     return 1;
}
```

Finally, main must enter an ‘event loop’ which detects and dispatches requests from (possibly remote) clients. This activity is specific to individual operating systems and ORBs and is discussed in the ORB specific sections below.