Information Engineering:
Object-Oriented Design and Analyses*

Chapter 2: Object-Oriented Programming and Databases

Steven A. Demurjian and Donald M. Needham
Computer Science and Engineering Department
The University of Connecticut
Storrs, Connecticut 06269-3155

January 13, 2006

Contents

1 The High-Tech Supermarket System ........................................... 3
2 Traditional Approaches to Design ........................................... 4
3 Design via Encapsulation and Hiding ....................................... 8
4 Object-Oriented Programming Concepts .................................. 12
  4.1 The C++ Programming Language ......................................... 13
  4.1.1 The Syntax/Semantics of C++ Classes .............................. 13
  4.1.2 The Role of the Constructor/Destructor ............................. 15
  4.1.3 Support for Hiding .................................................... 17
  4.1.4 Overloading ............................................................ 17
  4.1.5 The Static Storage Class ............................................. 18
  4.1.6 Two Kinds of Inheritance .......................................... 19
  4.1.7 Single vs. Multiple Inheritance .................................... 21
  4.1.8 Dispatching with Virtual Functions ................................ 22
  4.1.9 Friends (and Neighbors?) .......................................... 24
  4.1.10 Abstract Classes .................................................... 27
  4.1.11 Parameterized Types .............................................. 28
  4.2 The Ada95 Programming Language ..................................... 30
Software design techniques span a wide spectrum, and have incrementally evolved as the discipline has matured over the years. In the early 1960s, flowcharts were the most heavily used design technique for programming, and subsequently evolved through the sixties and into the mid-1970s into approaches such as data-flow and entity-relationship diagrams. At this same time, parallel efforts began on approaches for design using modules [26] and abstract data types (ADTs) [18, 19]. Module concepts were further explored in the late 1970s [35], taking us into the early 1980s, where these design concepts were supported in programming languages such as Smalltalk-80 [15], Ada [6], and Modula-2 [36]. Starting in the mid-1980s and continuing through today, there has been the emergence of the object-oriented approach for software design and development, with the majority of the research and development efforts focused on object-oriented programming languages and database systems. While it would be impossible to review this entire history in a single chapter, we will introduce and trace the important concepts that relate to: traditional approaches for software design, modules and ADTs, and, object-oriented programming, databases, and their integration.

The remainder of this chapter contains eight major sections. To serve as a basis for discussion in this and subsequent chapters, Section 1 introduces the High-Tech Supermarket System, HTSS. In Section 2, we review traditional approaches for software design, including: top-down, bottom-up, data-flow diagrams, and entity-relationship diagrams; highlighting their advantages and shortcomings. For brevity, the discussion is limited to these techniques; clearly there are others (e.g., finite state machines, Petri Nets, etc.) that could have been chosen (see also Exercise 1). Section 3 details techniques for encapsulation and hiding via modules and ADTs as a lead in to object-oriented programming using C++ and Ada 95 in Section 4, and object-oriented databases via Ontos and ODE in Section 5. Section 6 reviews an important article by Zdonik and Maier on object-oriented databases [37] that serves as the basis for our discussions in Section 7, on comparing and contrasting object-oriented programming and databases. Note that Sections 4, 5, 6, and 7 are loosely coupled – they are intended to provide a review of the important concepts and foundational material for the remainder of the textbook. Finally, in Section 8, we offer concluding remarks, a look ahead to Chapter 3, and related readings.

1 The High-Tech Supermarket System

The High-Tech Supermarket System, HTSS, is a modern supermarket that uses the newest and most up-to-date computing technology to support inventory control and to assist customers in their shopping experiences. The purpose of HTSS is to utilize computing technology in a positive way to enhance and facilitate the shopping experience for customers. The chain wants to integrate inventory control with:

1. the cashier’s functions for checking out customers to automatically update inventory when an item is sold,

2. a user-friendly grocery item locator that indicates textually and graphically where items are in the store and if the item is out of stock, and

3. a fast-track deli-orderer (deli orders are entered electronically), with the shoppers allowed to pick up the order weighed and packaged without waiting.

The inventory control aspect of the proposed system would maintain all inventory for the store and alert the appropriate store personnel whenever the amount of an item drops to its reorder limit. In addition to the aforementioned functional characteristics, the system should also have extensive query capabilities that allow store personnel to investigate the status of the inventory and
sales for the store over varying time periods and other restrictions. Finally, note that HTSS and its functional components are based on an actual store, Super Stop & Shop, that opened in Manchester, Connecticut, in the spring of 1993. Thus, the concepts that are presented have their basis in an actual ‘real-world’ application.

To support the functional and operational requirements of HTSS, from an end-user perspective, there must be a set of user-system interfaces. Possible interfaces include:

- **Cash Register/UPC Scanner**: Used to process an order, which includes: recording individual items, totaling them, deducting coupons, and taking payment. As each item is scanned, it must be deducted from the inventory, so that values are always consistent and up-to-date.

- **Displays for Inventory Querying**: To access and manage the inventory, a separate display is needed. Through this display, orders and updates can be made. Only authorized individuals will be allowed to enter new orders or update the inventory when a shipment arrives.

- **Shopper Interface for Locator**: Used by customers to locate where (aisle, shelf) a particular item is displayed in a store.

- **Shopper Interface for Orderer**: Through this interface, customers can place orders for the deli (e.g., meats, cheeses, salads, etc.). These orders are then filled and the customer picks up the order at some later time.

- **Deli Interface for Orderer**: This interface is needed by store employees that work in the Deli department to scan and fill customer orders.

We make no claims that these are the only user interfaces; rather they are the ones which are discussed in examples throughout this and later chapters. We have chosen this set based on both their differences (they all have unique requirements for their operation) and similarities (they all share common requirements regarding response-time, throughput, and user-friendliness). Response-time and throughput are important for the first two interfaces, since there is likely to be multiple cash registers that must work in parallel with many inventory displays. User-friendliness is also important, for new employees using cash registers, and especially for customers using the different shopper interfaces. Clearly, HTSS as an application, fits many of the requirements reviewed in Chapter 1: multiple types of data that must interact; performance constraints on throughput and number of concurrent users; persistence for multiple databases; and, a wide-variety of users with different capabilities and access requirements.

## 2 Traditional Approaches to Design

Traditional design approaches (e.g., top-down, bottom-up, data-flow diagrams, etc.) focus on developing a functional characterization of an application. Historically, there are close ties between these approaches and imperative or procedural programming languages like Fortran, Pascal, and C. The reason is that there is a direct correspondence between the design for an application using one of the approaches and its realization as a working piece of software or program, at both a conceptual-level and from the perspective of the coding and organizational techniques that are utilized to develop software using an imperative language. This section is a case study of a number of different traditional design approaches, namely, top-down, bottom-up, and data-flow and entity-relationship diagrams. Each approach can be used in many different ways, and is well suited to solving certain kinds of problems, or rather, to develop the solution to a problem from a specific perspective. Each approach can also be used to conceptualize a design at varying levels of granularity. Moreover, all of
the approaches suffer when one attempts to identify a certain stage or point in time that represents
the ‘complete’ or ‘finished’ design. Our purpose is to demonstrate these difficulties in the traditional
approach through a process that reviews and critiques each using the HTSS example.

Regardless of the traditional approach that is chosen, the common theme of a functional char-
acterization of the system or application pervades. In the case of HTSS, an enumeration of a subset
of the basic system functions would include:

1. Check-Out Customers: The actions that must be taken by a cashier to process a customer’s
order.

2. Locate Items: The actions that are taken whenever the location of specific items in the store
are necessary. This could be initiated as a result of a customer using the item locator user
interface or a bagger needing to check a price or get an item for a customer.

3. Order Deli Meats, Cheeses, and Salads: These are the actions that are necessary to support
the Deli Orderer, and to allow the order to be transferred to Deli employees for processing.

4. Update and Query Inventory: These actions are needed by management and stock-room per-
sonnel to track and maintain the status of the inventory.

These four major functions are determinable based on a top-down design examination of the problem
at hand. Once these functions have been identified, they can each be expressed as a set of detailed
tasks via the process of stepwise refinement. For HTSS, the first function can be refined as the
following tasks:

1a. Scan UPC Codes for all Items  
1b. Modify Inventory  
1c. Maintain Running Total  
1d. Subtotal/Coupon Adjustment  
1e. Final Total and Take Payment  

Tasks 1a, 1b, and 1c are interleaved to process all items for an order, followed by tasks 1d and 1e.
Each of these tasks can in turn be refined and expanded in an iterative and incremental process that
can be utilized to evolve the design towards an implementation. For example, as part of task 1e, if a
non-cash option is chosen, it may be necessary to verify credit card, ATM card, or checking account
status. This top-down process proceeds from the general to the specific to arrive at a solution.

The complement of top-down, is the bottom-up approach, which while still functionally-oriented,
is driven strongly by information and its usage. For example, given the four functions previously
reviewed (Check-Out Customers, Locate Items, Order Deli Meats, Cheeses, and Salads, and Update
and Query Inventory), and the general description of HTSS in Section 1, it is likely that a data
structure can be defined that maintains grocery items. In HTSS, each Item should have a UPC
(Universal Product Code) for unique identification, a Name, various Costs (e.g., Wholesale and
Retail), a Size or Weight, the Amount on shelves or in the stockroom, and so on. Given this
information, the major functional components of HTSS can be examined to identify their access
requirements. For example:

- UPC Scanner: Must scan the UPC on an Item, verify it against the database for the inventory,
  and then return all appropriate information on an Item to be used in checking-out a customer’s
  order.

- Locator: Once an Item has been selected by a customer, then the shelf Amounts can be accessed
to display quantity and location.
• **Inventory Control:** All of the responsibilities associated with managing the inventory which includes creating new entries for **Items**, updating existing entries, deleting **Items**, querying for both Scanner and Locator, and so on.

From these requirements, the commonalities can be identified and synthesized, to arrive at a set of procedures (and functions) that can support access to **Items**. For example, `Get_UPC_Code()` and `Get_Shelf_Amount()`, are two such functions. These low-level functions are used as building blocks to develop higher-level procedures and functions, which can then support the functional components of HTSS.

Whether top-down or bottom-up is utilized for design and implementation, there are still a number of important considerations that are not addressed by either approach. First, neither correctness nor completeness are part of the design process. As new refinements are made (top-down) or higher-level tasks are determined (bottom-up), there is no way to identify when we are done or whether the design matches the specification. Second, software evolution and extensibility are not explicitly supported by either approach. Third, while the bottom-up approach does achieve software reuse to a limited degree, the expansive process of stepwise refinement at times makes it very difficult to identify ‘common’ steps where code can eventually be reused. Fourth, both approaches seem counterproductive with respect to user-interface design, since they are prone to separate system functions from user interface needs. This can cause significant problems, particularly if a representation for a user interface is overlooked, which will result in the need to algorithmically construct the representation on-demand (costly in time and space). This also leads to user interfaces that are evolved rather than formally planned.

Another approach to design is data-flow diagrams (DFDs), which are used to describe system operations via a high-level characterization of information input/output, and the identification of major functional actions and informational flow. In the former, the emphasis is on what information must be input, stored, and displayed, so that it can be effectively used. In the latter, the focus is on how the information is used, by displays, individuals, other systems functions, and so on. DFDs as a design technique are very versatile. To represent high-level system behavior, a macroscopic view of an application, DFDs can characterize major system functions, as shown for HTSS in Figure 2.1. The major functional characteristics of HTSS are represented in a DFD using circles. Actions for input by a user or system are found in the rectangular boxes. Databases, or repositories of information, are indicated by the parallel lines (open boxes) which enclose a phrase. Displayed, or output information, is identified by the rectangular box with the upper right corner squiggled. The arrows indicate data or information flow. The five actions that are represented correspond to the five user-system interfaces presented for HTSS in Section 1 of this chapter.

DFDs can also be utilized to expand or explore a certain function of a system in greater detail. For HTSS, Figure 2.2 contains a DFD that might represent the Process Order function from Figure 2.1. There are three major tasks to process an order. First, each item must be scanned, recorded on the receipt, and updated in the inventory. Once all items have been processed, the second task subtotals and subtracts all appropriate coupons. The third and final task completes the order with payment by the customer and verification of valid credit by the cashier.

DFDs are still very popular today, since they are very easy to use, learn, and understand, even for individuals that do not have a computer science and engineering background. However, one important omission in DFDs is the inability to easily specify sequencing and iteration among the various tasks. Consequently, for the DFDs in Figures 2.1 and 2.2, the flow of control across the diagram is neither obvious nor inferable. Thus, DFDs clearly lack engineering rigor and discipline, and suffer the same completeness and correctness problems of the top-down and bottom-up approaches. We

---

1We have utilized concepts and notation for DFDs from [14].
are not alone in our assessment of these shortcomings. Ghezzi also notes that DFDs can only be considered as "... semi-formal notation", and must be used as part of a bigger picture where system structure not represented by DFDs can be captured by other techniques [14, pages 165-166]. We strongly agree with Ghezzi, and believe that DFDs, entity-relationship diagrams, and so on, can be used in conjunction with object-oriented techniques, as we will see in Chapter 3 and later chapters.

Since DFDs only support the representation of information at a coarse granularity level (i.e., large categories of information with little regard to its makeup and content), it can be complemented with entity-relationship diagrams for supporting a detailed conceptual modeling of the database requirements for an application. The entity-relationship (ER) approach was originally proposed by Chen in 1976 [12]. The basic building blocks of the ER approach are entities, to model static information aggregations, and relationships, to model static information associations. While the original ER approach did not contain the ability to specify inheritance among entities, later extensions have provided that modeling choice. Figure 2.3 contains an ER diagram for HTSS.

As a design technique, ER diagrams have many advantages. First, they are an excellent technique for conceptual database design that are easily utilized to represent information, as shown for HTSS in Figure 2.3. Second, both the information and its interdependencies can be identified and modeled. Third, by supporting inheritance, generalization is promoted to reduce information redundancies. Despite these advantages, there are also many drawbacks. First and foremost is the correctness/completeness issue, as we have discussed for earlier approaches. Second, by focusing on information and ignoring functional requirements and usage, it is very possible that one can arrive at an ER diagram that doesn’t meet the needs of the application. Third, ER diagrams lack the ability
of DFDs to represent interactions with other system components, e.g., user interfaces, systems functions, etc. This is critical for applications such as HTSS, where all of the various and diverse system components are interdependent and must work together. Finally, another major drawback is that from a design perspective, ER diagrams are really intended for use with database system. Issues that relate to integrating ER diagrams with programming languages have not been considered. Recall from Section 1.3 of chapter 1 that an important part of our approach was the presence of a database in future applications and the need to integrate programming and database perspectives.

3 Design via Encapsulation and Hiding

Module concepts for design were first proposed by Parnas [26] in the early 1970s, and realized in languages such as Modula-2 and Ada. As a concept, the motivation of modules can be tracked to the examination of existing programs that seemed to share similar solution approaches despite their different domains. Specifically, a given data type (e.g., record, structure, etc.) in a program always seemed to have a set of dedicated procedures and functions that represented its capabilities. Informally, this situation was often organized in separate files. Modules formalized this ad-hoc process by recognizing that most programs are partitionable into discrete program units, with well-defined interactions. Individually, each program unit encapsulated the required functionality for a given task, and provided an interface for a user, while hiding its implementation. Representation independence is achieved since changes can be made to the implementation that have no impact on the interface and its users. Collectively, the ability to encapsulate functionality and to support future changes was the driving force behind introducing modules as an improved design approach.

In many ways, design using modules mirrors the top-down approach of stepwise refinement. The process begins with the identification of major system tasks. These system tasks might correspond to information or operation. For HTSS, as shown in Figure 2.4, modules for system tasks such as Cashier, Scanner (used by Cashier, Locator, Orderer), and so on, are needed, as well as
modules for information such as **Items** and **DeliItems** (which are used by many other modules).

Thus, modules appear to encompass both DFD and ER approaches. Each task in a system will be realized by a module that contains a dedicated set of information and a set of procedures and/or functions that operate on the information to accomplish its task. A subset of a module’s information, procedures, and/or functions is identified or tagged for *export* to other modules. Exported portions of a module represent its interface to the ‘world’. Finally, since a module might interact with other modules, a module may *import* information and functionality for either standard actions (e.g., say, for I/O, printing, strings, etc.), or from other modules of the application. Notice that portions that are imported by one module must have been exported by other modules. Concepts for defining a module and importing (exporting) from (to) other modules, are also shown in Figure 2.4.

When using modules, it is advised that designers strive for low coupling and high cohesion. Low coupling implies that the interdependencies of modules with respect to exchanging information, is minimal. High cohesion refers to the ability of a module to characterize a single, well-defined task. Thus, through modules, controlled sharing is promoted, portions that are not exported from a module are hidden from other modules, and representation independence is facilitated. As we will see shortly, there are strong parallels between modules and ADTs. However, modules still suffer from the fact that there is no way to determine if the partitioning of the application into the resulting set

```plaintext
<table>
<thead>
<tr>
<th>Customer AcctNum</th>
<th>Order Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order</td>
<td></td>
</tr>
<tr>
<td>Deli Order</td>
<td></td>
</tr>
<tr>
<td>UPC Name</td>
<td>WCost Weight</td>
</tr>
<tr>
<td>RCost Item</td>
<td>ISA Deli CostLb</td>
</tr>
<tr>
<td>Size Item</td>
<td>Item Increm</td>
</tr>
<tr>
<td>OnShelf Location</td>
<td>Sales Daily</td>
</tr>
<tr>
<td>InStock</td>
<td>Total Sales</td>
</tr>
<tr>
<td>Location</td>
<td>Date</td>
</tr>
<tr>
<td>ROLimit</td>
<td></td>
</tr>
<tr>
<td>AcctNum Credit</td>
<td>Credit Info</td>
</tr>
<tr>
<td>Status Balance</td>
<td></td>
</tr>
<tr>
<td>ISA ISA ISA</td>
<td></td>
</tr>
<tr>
<td>Credit Card</td>
<td>Check Debit</td>
</tr>
<tr>
<td>Info Card</td>
<td>Card</td>
</tr>
<tr>
<td>Limit OvDraft</td>
<td>PIN</td>
</tr>
</tbody>
</table>
```

Figure 2.3: A ER Diagram for HTSS.
of modules is consistent, correct, or complete. In addition, while modules bridge, in part, the gap between functional behavior and information usage, their imperative approach still has problems related to reuse. Namely, reuse of modules and their association with different data types occurs at compile time; the runtime differentiation between different instances is not supported.

In Section 2 of Chapter 1, the basic or core abstract data type (ADT) concepts were presented. These issues are reconsidered with an emphasis on design. ADTs promote the design and development of applications from the perspective of information and its usage. From an information-perspective, there are ties between ADTs and the ER approach. From a usage-perspective, the functional characteristics can be explored in either a top-down or a bottom-up direction. However, ADTs take a combined view that focuses on information and its manipulation, which will yield a different design solution than an approach that considers each facet individually.

In the ADT design process, there are a number of considerations that must be addressed, as outlined below:

- Identify the Major Information Units: Determines the ADTs that are needed for an application or system.
- Describe the Purpose of Each Units: Indicates the overall responsibility for each ADT in the application.
- Define Manipulation Techniques for Each Unit: For ADTs, this corresponds to the operations or methods that must be characterized, including the signature.
Encapsulate and Hide: Representation of each unit and its manipulation are both encapsulated and hidden within the ADT.

In addition to describing the design for individual ADTs, we must also indicate the iterative or cyclical process that can be utilized to arrive at a design solution. We have taken a bottom-up approach to ADT design for HTSS, as shown in Figures 2.5 and 2.6. In Figure 2.5, the lowest-level of ADT is shown, with an emphasis on the information units for an application. Thus, we have ADTs for Item, DeliItem, Receipt, and so on. Given this lowest level, other ADTs can be designed that combine and utilize multiple low-level ADTs, as shown in Figure 2.6. In this case, the Process_Order ADT uses multiple ADTs, while the Sales_Info ADT only uses Receipt. Current and lower levels are incrementally combined to increase ADT functionality. Eventually, an ADT that describes the uppermost level of system behavior will be specified. Note that a top-down approach to ADTs is also reasonable and feasible.

```
ADT Item;
  PRIVATE DATA: SET OF Item(s), Each Item Contains:
    UPC, Name, wCost, RCost, OnShelf, InStock, Location, ROLimit;
    PTR TO Current_Item;
  PUBLIC OPS: Create_New_Item(UPC, ...) : RETURN Status;
    Get_Item_NameCost(UPC) : RETURN (STRING, REAL);
    Modify_Inventory(UPC, Delta) : RETURN Status ;
    Get_InStock_Amt(UPC) : RETURN INTEGER;
    Get_OnShelf_Amt(UPC) : RETURN INTEGER;
    Check_If_On_Shelf(UPC): RETURN BOOLEAN;
    Time_To_Reorder(UPC): RETURN BOOLEAN;
    Get_Item_Profit(UPC): RETURN REAL;
    Get_Item_Location(UPC): RETURN Location;
  ...
END Item;

ADT DeliItem; ADT CustomerInfo;
  PRIVATE DATA: SET OF (Item, Weight, ...
    CostLb, Increm); END CustomerInfo;
  ...
END DeliItem;

ADT Receipt; END Shelf_Info;
  PRIVATE DATA: SET OF Items;
    SET OF Coupons; {An ADT} ADT Sales_Info;
    SubTotal, Total, PayType; ...
  ...
END Receipt;
```

Figure 2.5: Low-Level ADTs for HTSS.

Clearly, the advantages of ADTs are similar to those of modules, with abstraction, reuse, and evolution promoted. However, there is still a problem related to completeness and correctness, since the techniques are somewhat ad-hoc, and decisions made at higher.levels (for the bottom-up approach) are impacted by lower-levels, i.e., if ADTs at the lowest level are wrong, those errors are carried through all subsequent levels. In addition, the lack of inheritance for ADTs will likely result in design redundancies, even though there is reuse. Finally, there are still difficulties when one attempts to represent dynamic system behavior (i.e., only-defined at runtime), or to establish
associations (other than inclusion) between ADTs. Both of these capabilities are critical to more accurately model the ‘real-world’ with ADTs.

```plaintext
ADT Process_Order; {Middle-Level ADT}
PRIVATE DATA: {Local variables to process an order}
PUBLUC OPS : {What do you think are appropriate?}

{This ADT uses the ADT/PUBLIC OPS from Item, Deli_Item, Receipt, Coupons, and Customer_Info to process and total an Order. Each Receipt must be cataloged and stored when an Order has been completed}
...
END Process_Order;

ADT Sales_Info; {Middle-Level ADT}
PRIVATE DATA: {Local variables to collate sales information}
PUBLUC OPS : {What do you think are appropriate?}

{This ADT uses the ADT/PUBLIC OPS from Receipt so that the sales information for the store can be maintained.}
...
END Sales_Info;

ADT Cashier; {High-Level ADT}
PRIVATE DATA: {Local variables used by a cashier}
PUBLUC OPS : {What do you think are appropriate?}

{This ADT uses the ADT/PUBLIC OPS from the middle-level ADTs (Process_Order, Sales_Info, etc.), and for the low-level ADTs.}
...
END Cashier;
```

Figure 2.6: Middle and High Level ADTs for HTSS.

4 Object-Oriented Programming Concepts

Despite the many drawbacks and disadvantages of traditional approaches, their historical place and long-term acceptance in computer science and engineering indicate that they will still play a role in current and future design. We will see throughout the course of the book that top-down and bottom-up techniques can be utilized to more precisely and accurately represent the features and characteristics of systems and applications. This is especially true for inheritance, since a top-down view promotes specialization, while a bottom-up view supports generalization, both of which are critical for effective design. DFDs and ER diagrams are also necessary, to allow the application/system to be examined from these complementary perspectives. Modules are a static version of ADTs, that support encapsulation and hiding, and while ADTs are a pure design technique, modules are available in programming languages such as Modula-2 and Ada. The questions that are posed at this point and addressed through the remainder of this chapter and book, respectively, are:

- Does the object-oriented paradigm embody all of the representational capabilities and advantages of top-down, bottom-up, DFD, ER, module, and ADT approaches?
• Can the object-oriented paradigm be extended and enhanced with capabilities, methods, and techniques that promote information engineering as discussed in Chapter 1?

In this section, we begin this process by examining the object-oriented approach via the C++ and Ada95 programming languages.

4.1 The C++ Programming Language

The purpose of this section is not to provide a comprehensive and complete introduction to C++, since there are many different books that provide these details [4, 27, 33]. Rather, a broad overview of C++ is provided, with an emphasis on the important object-oriented concepts that are supported. Also note that while different object-oriented programming languages have significantly different capabilities and functionalities, C++ appears to be the most popular of available languages. This is another reason that it has been chosen to introduce concepts; others including Smalltalk, Objective-C, ObjectPascal, Modula-3, Eiffel, CLOS, and Lisp Flavors, could also have been chosen. Finally, note that many compilers also have their own (non-standard) object-oriented extensions. To provide structure to the presentation, a total of 11 different subsections are utilized, with the HTSS serving as our continuing explanation vehicle.

4.1.1 The Syntax/Semantics of C++ Classes

The transition from procedural to object-oriented programming is best illustrated by examining data-structuring techniques. Typically, when using C for large-scale development, structures are defined in a header or ‘.h’ file, with a corresponding ‘.c’ or implementation file that contains the source code for all of the C functions that are needed to manipulate the structure. However, this is an informal technique that is left entirely as the responsibility of the software engineer and is relegated to the domain of good programming practice; it is not enforceable by the C compiler. Conceptually, classes in C++ are C structures with functions that are encapsulated, so that the ‘practice’ can be verified and insured by the compiler. Consider the C structure for Item in HTSS:
struct Item {
    int UPC, OnShelf, InStock;
    int ROLimit; // RO for Re-Order
    char* Name;
    float RCost, WCost; // R = Retail; W = Wholesale
    Loc Location; // Loc is another structure

    Status Create_New_Item(int UPC, ...);
    NameCost* Get_Item_NameCost(int UPC);
    void Modify_Inventory(int UPC, int Delta);
    int Get_InStock_Amt(int UPC);
    int Get_OnShelf_Amt(int UPC);
    Boolean Check_If_On_Shelf(int UPC);
    Boolean Time_To_Reorder(int UPC);
    float Get_Item_Profit(int UPC);
    Loc* Get_Item_Location(int UPC);
};

In C, the structure would only contain the data, with the functions placed in the implementation file. By including the functions as part of the structure, names and actions are specifically bound to Item, and thus, those functions are not usable with other data types. Given this structure, its access and use could be as follows:

    Status s;
    NameCost* nc;
    Item I1, I2;

    s = I1.Create_New_Item(12345, ...);
    s = I2.Create_New_Item(98765, ...);
    nc = I2.Get_Item_NameCost(98765);

In C, the dot notation, identifier.identifier, is used to select the different data members. In the above example, there must now be the selection of the correct function of the structure. Notice, that by including the functions as part of the structure, they can be more effectively encapsulated, than with the traditional C approach that simulates this feature via header and implementation files.

The transition from C structures with functions to C++ classes is primarily one of syntax, as shown below, where the Item class has been partitioned into a private implementation of data and a public interface of member functions:

class Item { // Header or ".h" File
    private:
        int UPC, OnShelf, InStock;
        int ROLimit; // RO for Re-Order
        char* Name;
        float RCost, WCost; // R = Retail; W = Wholesale
        Loc Location; // Loc is another class

    public: Item(); // The Constructor
        ~Item(); // The Destructor
        Status Create_New_Item(int UPC, ...);
        NameCost* Get_Item_NameCost();
        void Modify_Inventory(int Delta);
        int Get_InStock_Amt();
        int Get_OnShelf_Amt();

};
Like C, in C++, the class declaration is normally placed in a header file, with the implementation of the member functions placed in a separate implementation file. The difference is that in C++, the semantics of the specific class declarations are enforced in the implementation file and all other parts of the program that use the class. The definition of Item captures the characteristics that are shared by a group of objects or instances that are grocery items. However, like a record or structure, the Item class only represents a single occurrence or instance. If the desire is to model a group of Item instances, then another class (for sets, collections, etc.) would be necessary. This is similar to records in Pascal, where each represents a data type, and multiple records would be grouped as an array or list.

4.1.2 The Role of the Constructor/Destructor

In C++, for each class, a constructor and destructor must be defined. The constructor contains the allocation instructions to the compiler and is used when a particular variable is being declared (e.g., I1 in the previous section). Destructors are used to indicate the actions to be taken by the compiler when memory related to a variable must be freed (e.g., at the end of a function call, storage for local variables must be deallocated from the runtime stack). Constructors/destructors are not a new concept, since they are necessary for all system-defined types in a programming language (e.g., int, char, float, etc.). When a variable is declared in C via the declaration ‘int x[10]’; the constructors for both int and array must be invoked to correctly allocate memory. In C++, constructors/destructors are easily illustrated using the classic stack example:

```c++
class stack {
  private:
    char* st, top;
    int size;
  public:
    void stack(int x) {top = st = new char[x]; size = x;}
    ~stack() {delete st;}

    // Other public member functions for stack
};
```

```c++
main()
```
In this example, note that the constructor stack is given as a public member function that takes an integer as a parameter and initializes both the stack and its size. Thus, the two declarations for S1 and S2 create two character stacks with different sizes. The destructor for stack follows the constructor, must be given for all class declarations, and may be null. The destructor is invoked automatically throughout an executing program whenever deallocation is necessary (e.g., the end of a function call).

The approach for constructors given above is different from the Item class as discussed in Section 4.1.1. In the Item class, a constructor, Item(), and member function, Create_New_Item(int UPC, ...);, were given. The Item() constructor in this situation is used to create variables (allocate memory) at different points during the program, e.g., Item I1, I2; The Create_New_Item() member function is employed to initialize values for private data items at some point after their declaration. Essentially, the functionality of the constructor has been divided in two pieces, to both allocate memory (independent of the application) and to initialize private data items (dependent on user or system input).

In a different implementation, multiple constructors for Item may be defined, as given in:

```cpp
class Item // Declaration in "h" File
{
    private:
        int UPC; char* Name;
        int InStock, OnShelf, ROLimit;
        float RetailCost;
    public:  // Two constructors for versatility
        Item(int code, char* str, int st1,
             int st2, int st3, float cost);
        Item(); // Therefore, Constructor is overloaded!
        void CreateNewItem();
    // Remaining Member Functions Omitted
};

// Implementation in "c" File
Item::Item (int code, char* str, int st1,
            int st2, int st3, float cost)
{ UPC = code;
    Name = new char[sizeof(str)];
    strcpy(Name, str);
    InStock = st1; OnShelf = st2;
    ROLimit = st3; RetailCost = cost; }

Item::Item()
{ Name = new char[20]; // Default allocation }

// Use in "main.c" File
Item actual_item; // Allocates memory for an Item
Item *iptr = new Item(555, "Beans", 100, 30, 30, 0.39)
```

In this situation, the Item::Item() constructor is a default, and simply allocates memory to hold the Name. The use of this constructor is illustrated through the variable declaration Item actual_item;
To use the other constructor in a variable declaration, a software engineer would be required to provide initializing values for each parameter. This is shown with the iptr declaration, which utilizes new to allocate memory. In this case, the signature of this constructor must be known, so that the user can provide the appropriate initializing values for its invocation. Note that in this example, the constructor has been overloaded.

4.1.3 Support for Hiding

Hiding in C++ is supported by clearly identifying the private and public portions of a C++ class declaration, as shown in the previous example for Item. Each portion of a class can contain data and/or member functions. However, it is typical that the private portion contains data, with the public portion composed of the set of functions to manipulate the data. The public portion is that part of a class that is available to other classes (and users). The private portion of a class can only be accessed directly within the implementation of the member functions of a class, e.g., within Create_New_Item(int UPC, ...), Get_Item_NameCost(), and so on. Thus, the following code:

```c++
#include "item.h"
#include "stream.h"
main();
{
    Item I1;
    Status s;

    s = I1.Create_New_Item(12345, RiceCereal, ...);
    cout << I1.UPC; // Print out UPC for I1
}
```

would result in the generation of a syntax error, since there is an attempt to print out the UPC of I1. Overall, when a piece of data or member function is placed in the private portion of a class declaration, it becomes invisible to the outside world, effectively hiding the implementation. On the other hand, once a member function or data item is placed in the public portion of a class, it becomes accessible to any and all users of the class. Thus, the public interface can be characterized as the union of all possible functions for all potential users. A member function intended for one or two specific users is therefore available to all users. Therefore, it is critical that great care is taken when specifying the public interface. This is a topic which will be revisited a number of times in coming chapters.

4.1.4 Overloading

As discussed in Section 3.1 of Chapter 1, overloading has long been used by programming languages to reuse different operations across many data types. For example, in Pascal, the + operator is used for integer addition, for real addition, and for set union. In C++, operations and member functions can be overloaded. If we continue with the stack example from the previous section, we can overload the constructor as follows:

```c++
class stack {
    private:
        char* st, top;
        int size;
    public:
        void stack(int x) {top = st = new char[x]; size = x;}
        stack() {top = st = new char[100]; size = 100;}
        ~stack() {delete st;}
        // Other public member functions for stack
```
Notice that the first stack constructor is as given in Section 4.1.2. The second constructor can be thought of as a default, since it can be used to automatically initialize a stack when a size hasn’t been provided. The variable declarations for S1 and S2 would call the first and second constructors, respectively.

In C++, any operation may be overloaded, as long as there are differences between the signatures of the member functions in question. The difference must be significant enough to distinguish the functions (i.e., the return type is not sufficient), and must involve the parameters and/or their types. This was true for the two constructors of stack. Other useful operations for overloading include: ≪, ≫ for input/output using C++ stream I/O; ==, >, ≥, etc., for conditional comparison; and, =, +, −, etc., for assignment and algebraic expressions. For example, overloading auto-decrement (--) would be useful in HTSS to allow the -- to affect two private data fields, as given in:

```cpp
class Item { // Header or ".h" File
private:
    int    UPC, OnShelf, InStock, ROLimit;
    // Etc... As Previously Given

public:     Item(); // The Constructor
           // Etc... Others as Previously Given

    Item operator--(){ this->OnShelf--;  
                      this->InStock--; }

};
```

// The main() Function
#include "item.h"
#include "stream.h"
main();
{
    Item I1;
    Status s;

    s = I1.Create_New_Item(123, OJ, 10, 30, ...);
    I1--; // Reduces I1.OnShelf and I1.InStock
    // Now Contain 9 and 29 Respectively
}

In this case, I1-- changes two variables, since recording an item as sold must affect both its on-the-shelf and stock-room totals. Through overloading, software engineers are afforded a consistent view when using both system and application-defined classes.

### 4.1.5 The Static Storage Class

One interesting feature of C++, that is also used by other object-oriented programming and database languages, is the static storage class. In C, a variable that is prefaced by `static` results in a compiler instruction to allocate data for the variable at compile time rather than at runtime. Consider the C program:

```c

```
main();
{
    how_static_works();
    how_static_works();
}

how_static_works()
{
    static int x = 4;
    printf("The Value of x is: %i \n", x);
    printf("The incremented value of x is: %i \n", x++);
}

Notice that the integer variable x has been defined to be static. If x was just a local variable in a function, then its memory would be allocated off the runtime stack when the procedure is called. However, since it is not, storage is allocated at compile time, thereby allowing x to retain its value across multiple calls, like COMMON in Fortran. For the example, the first call to how_static_works would initialize x to 4, print its value, and then increment. The second call would print the value, now 5, and then increment.

In C++, the storage class static has a significantly different interpretation. Specifically, static indicates that the data member of the class is shared by all instances of the class. In this situation, static allows software engineers to define class variables (shared by all instances) as opposed to instance variables (specific to each instance). For example, consider a class Sales_Info for HTSS:

class Sales_Info {
private:
    static int total_orders = 0;
public:
    inline void Increment_Total() { total_orders++; }
    // Other public member functions for stack
};

main()
{
    Sales_Info sale_1, sale_2;
    sale_1.Increment_Total();
    sale_2.Print_Total_Orders();
}

In this example, the integer variable total_orders is defined to be static and initialized to zero. When the two variables sale_1 and sale_2 are declared in the main program, despite the fact that they represent different instances of Sales_Info, they share the same storage for the total_orders variable. Thus, if sale_1 increments the total, then sale_2 will print the incremented value.

4.1.6 Two Kinds of Inheritance

Inheritance, as a concept, allows information and behavior to be passed from one class (called the super, parent, or base class) to another class (called the sub-class, child, or derived class). In HTSS, inheritance is very useful, since commonalities can be identified and grouped. This was shown in the ER diagram of Figure 2.3, where DeliItem was a subclass of Item, since it needed the many features of Item, but required additional private data to keep track of the cost per pound and the increment of sale. Other sub-classes of Item for produce, baked goods, meats, etc., would also be appropriate.
C++ supports two kinds of inheritance, that are differentiated by the portions of the super class that are visible to descendants of the sub-class. The differences are traced to the public and protected portions of a C++ class. Like the private portion of a class, the protected portion is still invisible to the outside world. However, the protected data and member functions can be passed from a base class to its derived class. Consider the Item class, reformulated as:

```cpp
class Item {
    private: // Typically Not Used
    protected:
        int UPC, OnShelf, InStock;
        int ROLimit; // RO for Re-Order
        char* Name;
        float RCost, WCost; // R - Retail; W - Wholesale
        Loc Location; // Loc is another class

    public: // As Before
};
main()
{
    Item I1;
    I1.UPC = 12345; // Still causes a syntax error!
}
```

where the formerly private data is now protected, but still unaccessible! This example also illustrates a 'practice' of software engineering for object-oriented development in C++, namely, the use of only the public and protected portions of a class declaration. This allows information to be more easily passed to sub-classes, without making the information visible to the outside world. The idea is that a sub-class, by its unique relationship, should be able to access and/or modify private data directly.

Given these new concepts, public derived inheritance allows both the public and protected portions of a class to be passed from a super class to a sub-class, which in turn may also be passed on to other descendants. In the example below, DeliItem is a public derived class of Item. Thus, what was protected (public) in Item is also protected (public) in DeliItem.

```cpp
class Item {
    protected:
        int UPC, OnShelf, InStock;
        int ROLimit; // RO for Re-Order
        char* Name;
        float RCost, WCost; // R - Retail; W - Wholesale
        Loc Location; // Loc is another class

    public: // As Before
};
class DeliItem : public Item {
    protected:
        float Weight, CostLb, Increm;
    public:
        DeliItem();
        ~DeliItem();
        float Profit_Per_Lb() {return RCost - WCost;}
};
```

However, if there was any private portion of the Item class, that portion would be unavailable for use by DeliItem. Notice also that the member functions of DeliItem can access the protected data of Item.
The second kind of inheritance, *private derived*, changes the characteristics of the public and protected portions of a class after the inheritance. That is, what was public and protected in the super class is still passed to the sub-class, but, upon reaching the sub-class, both the public and protected information turns private, and cannot be made explicitly accessible to any descendants of the sub-class. For HTSS, consider an example with classes for *Item*, *DeliItem*, and *SaladItem*:

```cpp
class Item {
    // As Before
};

class DeliItem : private Item {
    protected:
        float Weight, CostLb, Increm;
    public:
        DeliItem();
        ~DeliItem();
        float Profit_Per_Lb() {return RCost - WCost;}
};

class SaladItem : public DeliItem {
    protected: // Appropriate Protected Data
    public:
        float Profit_Per_Lb() {return RCost - WCost;}
};
```

In this case, the `RCost` and `WCost` variables that were protected in *Item*, are now private in *DeliItem*. Thus, the `Profit_Per_Lb` member function in *SaladItem* will result in a compilation error.

Overall, both of these kinds of inheritance are very important when developing class libraries that will be made available for general use. In such a situation, the software engineer designing the class library can use public derived inheritance for all internal nodes of an inheritance hierarchy. This allows the engineer to more easily share information. For the leaf classes in a hierarchy, private derived inheritance can be used, which still allows the protected portion to be accessed, but makes this same portion unavailable for use by others who will derive their own classes from the leaf nodes. This is a realistic scenario, since it is likely that only the leaf classes are the ones which a software engineer would want to make public in a class library.

### 4.1.7 Single vs. Multiple Inheritance

The discussion of Section 4.1.6 only focused on single inheritance, where any class has exactly one parent. Multiple inheritance expands this concept by allowing a class to inherit from more than one parent, when necessary. Multiple inheritance is acknowledging the simple fact that the world and its information does not occur in nice, easy to structure, hierarchies that are strict trees. Rather, there are many situations where the informational and/or operational requirements and characteristics of a class require it to inherit from more than one parent, thereby forming an inheritance graph or lattice. Thus, the major benefit of multiple inheritance is a more accurate conceptualization of the application, since these special and unique situations can be modeled.

In HTSS, multiple inheritance can be used to more effectively indicate the relationships among different *Items*. For instance, consider the inheritance hierarchy:

```
Item

  ProduceItem   MeatItem   BakeryItem   PharmItem
```
DeliItem

In this situation, the common parent Item has two purposes. First, it allows us to abstract out all data/member functions that are common to items, regardless of their types. Second, when necessary, we can treat all of the instances, regardless of their actual type, as Items. As we will see in the rest of Section 4, this provides us with powerful information structuring and programming capabilities.

However, multiple inheritance does cause many different problems, at both conceptual and implementation perspectives. First, there is the possibility of naming conflicts, both for data portions and member functions, since a class may inherit from separate paths in the hierarchy. For example, in HTSS, ProduceItem and MeatItem might both have data that tracks when to remove the items from the shelve. However, for ProduceItem these values might be up to one week, while for MeatItem, days are more typical. Second, a common node might be inherited more than once, which is true in the example since DeliItem can acquire Item characteristics either from ProduceItem or MeatItem. This information must only be inherited once by DeliItem, which is further complicated by the fact that ProduceItem or MeatItem must share the same and correct Item instance that comprises the core of DeliItem. Third, naming conflicts and the inheritance of a common node more than one time cause practical problems associated with a constructor, which must be more complex to handle these situations. This results in a runtime environment that is more difficult to manage and control (from the perspective of the compiler), and understand (by software engineers).

Putting the problems aside, multiple inheritance in C++ is defined in a similar fashion to single inheritance. For the previous hierarchy, we might have:

```cpp
class Item {
    // As Before
};

class MeatItem {
    protected:
        Date Expires; // Date When Meat Expires
    // Other, Appropriate Protected Data
    public:
        // Appropriate Public Member Functions
};

class DeliItem : public MeatItem, public ProduceItem {
    // Normal Class Declaration
};
```

Details for supporting multiple inheritance are very language and compiler dependent. The reader is referred to appropriate language reference manuals for supporting this important concept at an implementation level.

### 4.1.8 Dispatching with Virtual Functions

Dispatching is one of the most important features that is offered by the object-oriented approach, and is defined as the runtime or dynamic choice of the member function that is to be called based on the class type of the invoking instance. The concept is particularly useful in the presence of inheritance, since only in these situations can an instance be interpreted in many different ways, e.g., a T-bone steak (instance of MeatItem) can be also viewed as an instance of Item. Interpreting a group of instances as sharing the type of a common parent is important, since it allows us to cross
class boundaries to aggregate different instances when necessary. This is analogous to the ability to construct a list in C where the different list entries have different types. Such a scenario would be needed for HTSS, if the entire inventory was to be scanned, rather than just a search for one item of a particular type. Consequently, dispatching offers many different benefits to software engineering, all related to extensibility and productivity, including: more versatility in the development of inheritance hierarchies and class libraries; the promotion of software reuse and evolution; and, the ability to more easily and effectively develop and debug generic code.

To support dispatching in C++, a number of concepts are needed: the ability to support inheritance via derived classes (see Section 4.1.6 again); the redefinition of member functions in the derived class, which is a more advanced form of overloading (see Section 4.1.4 again); the utilization of virtual member functions in the parent (and other ancestors), as we will discuss shortly; and, the use of friends for designing and implementing a mixed-type collection of instances (see Section 4.1.9). To serve as a basis for the discussion, consider the class declarations:

```cpp
class Item { // As Before
    virtual void Print_Item_Info();
};

class DeliItem : public Item {
    protected:
        float Weight, CostLb, Increm;
    public:
        void Print_Item_Info()
            { cout << Weight << CostLb << Increm;
            Item::Print_Item_Info(); }
};

class ProduceItem : public Item {
    protected: int TempReq; // Temperature in Degrees F
    public:
        void Print_Item_Info()
            { cout << TempReq; Item::Print_Item_Info(); }
};

class BakeryItem : public Item {
    // Does Not Have Print_Item_Info Function
};

main()
{
    DeliItem D1; ProduceItem P1; BakeryItem B1;
    D1.Print_Item_Info();
    P1.Print_Item_Info();
    B1.Print_Item_Info();
}
```

In the example, the function `Print_Item_Info` is defined as virtual in the `Item` class, which indicates that it may be defined in sub-classes. A virtual function declaration is also an alert to the runtime environment that mandates that when the function is invoked, the sub-class is checked for the member function before the super class. If the member function is not found, then the super class is checked, and so on. The compiler guarantees that the member function must be defined somewhere along the path from the class of interest through its ancestors. If not, there would have been a compiler error. In the example, for the `D1` and `P1` instances, `Print_Item_Info` from their respective
classes is called. For B1, the virtual version in Item is invoked. This determination is made at runtime via the process of dynamic binding.

4.1.9 Friends (and Neighbors?)

While dispatching and virtual functions do provide some interesting capabilities, it is not until the friend concept is added that more advanced object-oriented programming features in C++ can be explored. The friend concept in C++ breaks encapsulation boundaries by allowing a class to selectively make portions of its private and protected portions available to other classes. This is in addition to the fact that via public derived inheritance, the protected portion of a super class can be acquired by a sub-class. When viewed from a perspective of programming-language design, friends are not a desirable feature, since they violate the precepts and principles of the object-oriented paradigm. However, in its implementation in C++, the friend construct requires that privileges must be granted by the designer/creator of a class. That is, for class A to be a friend of class B, A must have the statement “friend B” in its class declaration. Thus, there is some control in granting access to private and protected portions of a class. This is illustrated in the following:

```cpp
class Item {
    // As Before
    friend class DeliItem;
};
class DeliItem {
    // As Before
    float Profit_Per_Lb() {return RCost - WCost;}
};
```

Notice that unlike earlier examples, DeliItem does not inherit from Item. Despite this fact, the protected data of Item is now accessible, and therefore no syntax error will be generated for the Profit_Per_Lb function definition. The use of friend can occur at fine (one portion of data or member function) or coarse (the entire private or protected portion of a class) levels of granularity.

When friends are used in conjunction with virtual functions, the true power and capabilities of dispatching can be demonstrated. In the example below, a subset of the inheritance hierarchy for HTSS is given, with virtual functions defined in each class for both inserting a new item and printing information of an existing item. Notice that the Item class (common parent) contains a pointer (first_ptr) to a list of all grocery items (next_item).

```cpp
class Item {
    protected:
        static Item* first_ptr; // Refers to initial Item
        Item* next_item; // Next Item on a list
    public: // Other Details As Before
        virtual void Print_Item_Info();
        virtual void Insert_Item() { ... }
    friend class ItemCollection;
};
class DeliItem : public Item {
    public: // Other Details As Before
        virtual void Print_Item_Info() { ... }
        virtual void Insert_Item() { ... }
    friend class ItemCollection;
};
class SaladItem : public DeliItem {
```

```cpp
24
public: // Other Details As Before
   virtual void Print_Item_Info() { ... }
   virtual void Insert_Item() { ... }
friend class ItemCollection;
};

class ProduceItem : public Item {
   public: // Other Details As Before
      virtual void Print_Item_Info() { ... }
      virtual void Insert_Item() { ... }
friend class ItemCollection;
};

Notice that each Insert_Item member function is redefined in each sub-class, and its implementation (not shown) will call Insert_Item of its immediate parent. The actual calling sequence in this situation is determined at run-time based on the type of the invoking instance. In addition, each class is a friend of the new class ItemCollection, which will be used to represent a set of one or more Items, regardless of their underlying types. The class declaration and main program are:

class ItemCollection {
   protected: // Appropriate Protected Data to
              // Init and Manage a Collection
      Item* HeadPtr;
public:
   // Other functions have been omitted
      void Print_All_Items() {
         Item* ItemPtr = HeadPtr;
         while (ItemPtr != NULL)
            { ItemPtr->Print_Item_Info();
              ItemPtr = ItemPtr->NextItem; } }
      void Insert_An_Item(Item* i_ptr) { ...
         Item* ItemPtr = HeadPtr;
         i_ptr->Insert_Item();
         i_ptr->first_ptr->next_item = ...;
         HeadPtr = i_ptr->first_ptr; ... }
};

main()
{
   ItemCollection StoreItems;
   SaladItem S1(11111, Potato, ...);
   DeliItem D1(22222, Bologna, ...);
   ProduceItem P1(33333, Celery, ...);

   StoreItems.Insert_An_Item(&S1);
   StoreItems.Insert_An_Item(&D1);
   StoreItems.Insert_An_Item(&P1);

   StoreItems.Print_All_Items();
}

Notice that in ItemCollection, the pointer and its manipulation are all with respect to Item, since this is the common parent shared by all of the classes in the inheritance hierarchy. Also note that the two member functions of ItemCollection that are given, Print_All_Items and Insert_An_Item,
call \texttt{Print\_Item\_Info} and \texttt{Insert\_Item}, respectively, which were defined for \texttt{Item} and its sub-classes. Further, note that the reason \texttt{ItemCollection} must be a friend is that the protected data of \texttt{Item} is changed, namely, \texttt{first\_ptr} and \texttt{next\_item}.

Given this scenario, the execution of the first four statements of the main program would create variables as follows:

\begin{verbatim}
+--- +----------+ +----------+ +-----------+ 
StoreItems | | S1 | SaladItem| D1 | DeliItem | P1 |ProduceItem|
+--- | | | | | | |
| 11111 | | 22222 | | 33333 |
| Potato | | Bologna | | Celery |
| Salad | | ... | | ... |
| ... | | | | |
+----------+ +----------+ +-----------+ 
\end{verbatim}

Initially, the \texttt{StoreItems} variable that will represent the collection is empty. After the declarations, the next three calls to \texttt{Insert\_An\_Item} are made to establish the collection. Each of these calls also calls the individual \texttt{Insert\_Item} functions that are defined in each class, which is allowed due to friend privileges. As each item is placed in the collection, its put in as an \texttt{Item}, the common class, rather than as a specific \texttt{SaladItem}, \texttt{DeliItem}, or \texttt{ProduceItem}. Dispatching occurs at runtime. The execution environment while recognizing all of these three instances as being of type \texttt{Item}, will also examine the underlying type (\texttt{SaladItem}, \texttt{DeliItem}, or \texttt{ProduceItem}) to call the correct \texttt{Insert\_Item} in the subclass. After the inserts have occurred, the result is:

\begin{verbatim}
StoreItems:

+-----------------------------------------------+-----+-----+-----|
| Item | Item | Item |
| HeadPtr ---|---|---|---|
| (Really | (Really | (Really |
| | | | |
| Potato | Bologna | Celery |
| Salad | | | |
| | | | |
| +-----+-----+-----|
\end{verbatim}

At this point, when \texttt{Print\_All\_Items} is called, the collection is traversed. This traversal is possible, since from the perspective of \texttt{StoreItems}, all instances of the collection share a common type, namely, \texttt{Item}. Thus, the call to \texttt{Print\_Item\_Info} would call the functions in \texttt{SaladItem}, \texttt{DeliItem}, and \texttt{ProduceItem}, respectively. With dispatching, it is possible to treat a set of instances as the same, if they share a common parent, when it’s convenient and suits the needs of the application. Despite this common grouping, the underlying type of the instances can be automatically used at runtime to realize a specific behavior.

The major benefit and real power of dispatching relates to extensibility and software evolution. Suppose for HTSS that a new sub-class, \texttt{BakeryItem}, was defined, with a similar structure to other sub-classes of \texttt{Item}, including that it’s a friend of \texttt{ItemCollection}. Clearly, the main program given above would change to the following:

\begin{verbatim}
main()
{
  ItemCollection StoreItems;
  SaladItem  S1(11111, Potato, ...);
  DeliItem   D1(22222, Bologna, ...);
  ProduceItem P1(33333, Celery, ...);
  BakeryItem B1(44444, Bread, ...);
}\end{verbatim}
StoreItems.Insert_An_Item(&S1);
StoreItems.Insert_An_Item(&D1);
StoreItems.Insert_An_Item(&P1);
StoreItems.Insert_An_Item(&B1);

StoreItems.Print_All_Items(); }

While a new class for BakeryItem must be defined, and a recompilation is necessary, there are no changes to the ItemCollection class. The StoreItems.Print_All_Items statement will work correctly; and consider BakeryItem as valid. Thus, the extension is seamless and easily done with a minimal affect to the existing implementation. Note also that sub-classes of SaladItem or MeatItem could be added in a similar fashion without the need to make significant changes.

4.1.10 Abstract Classes

In C++ and most other object-oriented languages, it is possible to declare what is known as an abstract class. An abstract class is a base class (i.e., typically the root) in C++ which can never be instantiated. An abstract class is defined if at least one of its member functions has a null definition; such a function is called a pure virtual function. Item could be reformulated as an abstract class as follows:

```cpp
class Item {
  private:
    int UPC, OnShelf, InStock;
    int ROLimit; // RO for Re-Order
    char* Name;
    float RCost, WCost; // R - Retail; W - Wholesale
    Loc Location; // Loc is another class

  public:
    Item(); // The Constructor
    ~Item(); // The Destructor
    virtual Status Create_New_Item(int UPC, ...) = 0;
    virtual NameCost* Get_Item_NameCost() = 0;
    virtual void Modify_Inventory(int Delta) = 0;
    virtual int Get_OnShelf_Amt() = 0;
    // ...Etc... for other member functions
};
```

The sub-classes of Item would still be definable as discussed earlier. However, it would no longer be possible to create an instance of type Item; only its descendants are instantiatable. If there is at least one pure virtual function in a class, the compiler automatically casts it as an abstract class and prohibits instantiation.

Such a facility is very useful in object-oriented programming, and this concept may have its roots in database systems and semantic data models. For example, in the functional data model and Daplex [30], which is similar to the ER approach, instantiation is only allowed at the leaf nodes of inheritance hierarchies. This is a reasonable restriction. Suppose we assume a class hierarchy with Person as the root, Student and Faculty its sub-classes, and Undergrad and Graduate as subclasses of Student. In Daplex, only instances for Undergrad, Graduate, and Faculty can be created. This is saying that a Student instance can’t exist singularly, but only as part of Undergrad and Graduate instances. A similar restriction holds for Person instances. In some applications this is necessary, since it eliminates dangling instances, and controls the allowable situations.

27
4.1.11 Parameterized Types

The final, important object-oriented concept which was presented in Section 3.1 of Chapter 1 is parameterized types, which supports genericity and promotes class reuse. A parameterized type is another term for a generic, and simply refers to a segment of code that can be applied to multiple data types, since it contains a parameter for type information. In C++, the template feature has been proposed to support parameterized types, and is illustrated by reverting back to the stack example:
template <class T> class stack {
    private:
        T* st;
        int top;
        int size;

    public:
        void stack(int x) {st = new T[x]; top = 0; size = x;}
        stack() {st = new T[100]; top = 0; size = 100;}
        ~stack() {delete st;}
        push(T entry) { st[top++] = entry;}

        // Other member functions as needed
};

main()
{
    stack<int> S1(10);
    stack<char> S2();
}

Notice in the example that the type serves as a parameter to the stack in the call to the constructor in the variable declaration. This type parameter is substituted for all occurrences of T in the example. When using templates in C++, both the declaration ('.h' file) and the implementation ('.c' file, not shown in the example) must be combined into a single file.

We are not limited to simple types; a declaration such as stack<Item> StoreItems; for the HTSS application is also possible. In fact, it is also possible to have a template use another template in its class definition. In the example below, a template for a Collection has been defined, which uses a doubly-linked list as a means for storing the members of the collection.

template <class CollMemberType> class Coll {
    DList<CollMemberType>* _members;
    public:
        Coll() { _members = new DList<CollMemberType>(); }
        void Add(CollMemberType* member) { _members->queue(member);}
        void Remove(CollMemberType* member) { _members->remove(member);}
        int Member(CollMemberType* member) {return _members->isMember(member);}
        int NullColl(){return _members->isEmpty();}
};

// Use of Coll Template
main()
{
    Coll<int> IntColl;
    Coll<Item> ItemDB;
}

The interesting portion of this example is found in the use of DDList, which is a template declaration for a doubly-linked list. DDList contains member functions for queue, to add an element, and remove, isMember, and isEmpty with obvious interpretations. DDList could have been developed for another application, and as such, was likely tested extensively. This means that there is a very high probability that there are neither syntax nor semantic errors in the implementation for DDList. This in turn means that designing, implementing, and testing Coll will be facilitated by the strong likelihood that DDList is error free. This is the real strength of templates; the facilitation of reuse that naturally leads to a reduction in design and development.
4.2 The Ada95 Programming Language

Like the previous section on C++, it is not our intent to provide a comprehensive discussion of Ada95 [7], which has extended Ada83 [9, 22] with object-oriented capabilities. Rather, this section mirrors Section 4.1.1 by reviewing the major object-oriented capabilities of Ada95 using essentially the same examples. This allows the reader to compare and contrast the two programming languages based on a common ground.

4.2.1 The Syntax/Semantics of Ada 95 Classes

The transition from procedural to object-oriented programming is best illustrated by examining data-structuring techniques. Typically, when using Ada 83 for large-scale development, types (generally records) are defined in a package specification (analogous to a ‘‘.h’’ file in C/C++) along with the profiles of the procedures/functions that a client package can use to manipulate the data fields of the record. A corresponding implementation file, called the package body (analogous to a ‘‘.c’’ file in C/C++) contains the source code for all of the procedures/functions that are needed to manipulate the record. This separation of interface from implementation provides a solid vehicle for data abstraction, thereby supporting encapsulation and representation independence. Missing from Ada 83 is the ability to extend a type with both record fields and operations, and the concept of polymorphism.

In Ada 95, extensible records, referred to as tagged types, similar to the classes in other object-oriented languages, form the basis for object-oriented programming. Fields declared within the tagged record are referred to as components (data members). Procedures/functions that are declared in the same package specification as a tagged type or have the tagged type as a parameter or result are referred to as primitive operations (methods). Consider the Item tagged type for HTSS as given in the following package definition:

```ada
package Item_Package is
  type Item is tagged private; -- An extensible type

  function Create_New_Item(An_Item : in out Item; UPC : integer, ...);
  function Get_Item_NameCost(An_Item: in out Item) return NameCost;
  procedure Modify_Inventory(An_Item : in out Item; Delta : integer);
  function Check_If_On_Shelf(An_Item: in out Item) return boolean;
  function Time_To_Reorder(An_Item : in out Item) return boolean;
  function Get_Item_Profit(An_Item : in out Item) return float;
  function Get_Item_Location(An_Item: in out Item) return Loc;

  private
    type Item is tagged
      record
        UPC, OnShelf, InStock, ROLimit : integer;
        RCost, WCost : float;
        Name : Name_Type; -- Name is another record
        Loc : Location_Type; -- Loc is another record
      end record;
  end Item_Package;
```

Note that the above package definition would be placed in one file, with its implementation of all of the procedures/functions, called the package body (not shown), placed in a separate file. Once compiled, the package’s primitive operations can be used as follows:

```ada
nc : namecost;
item1: Item;
```
The primitive operation `Create_New_Item` is invoked on the `item1` object by passing `Create_New_Item` the `item1` object, which may include other parameters. The primitive operation `Get_Item_NameCost` is invoked as a function which returns a value of type `namecost` when passed an object of type `Item`.

Like Ada 83, in Ada 95, a type declaration is normally placed in a package specification, with the implementation of the primitive operations in a separate file containing the package body. The definition of `Item` captures the characteristics that are shared by a group of objects or instances that are grocery items. However, like a record, the `Item` tagged type only represents a single occurrence or instance. This is illustrated by examining the implementation of the tagged type `Item` in its package body:

```ada
package body Item_Package is
    procedure Create_New_Item(An_Item : in out Item; UPC : integer; ...) is
        begin
            -- Code for Loading Instance Component Values
            end Create_New_Item;
            -- Code for all Other Member Functions of Item
end Item_Package;
```

If the desire is to model a group of `Item`'s instances, then another tagged type (for a set, collection, etc.) would be necessary. This is similar to records in Pascal or structures in C, where each represents a data type, and multiple records would be grouped as an array or list.

### 4.2.2 Constructing Objects With Initial Component Values

Ada 95 objects are created using a tagged type definition similar to system-defined types (e.g., integer, float, etc.). Ada 95 allows more generalized use of a type via discriminants. Discriminants are parameters to a type that provide the information that is needed to construct the desired object (instance). Objects declared using tagged types with discriminants are required to provide values for all discriminants. To illustrate the use of discriminants, consider the classic stack example:

```ada
package Stack_Package is
    type stack (Size : integer) is private;
    procedure push(S : in out stack; X : in out character);
    ...
    private
        type char_vector is array (integer range <> ) of character;
        type stack (Size : integer) is
            record
                S : char_vector(1..Size);
                Top : integer range 0..Size := 0;
            end record;
    end stack;
end Stack_Package;

procedure two_stacks is
    stack1 : stack(10);
    stack2 : stack(20);
    -- Creates Two Different Size Character Stacks!!
```

In the example, the parameter `Size` is a discriminant. A nuance of Ada 95 is that discriminants may NOT have default values if inheritance is to be used. In effect, all objects are required to provide values for all discriminants upon declaration. Within the package where the discriminants
are declared, the discriminants are treated like ordinary components of the type, with one notable exception, they can’t be changed once they are defined (i.e., discriminants are held constant).

For HTSS, a version of the tagged type `Item` that uses discriminants is given below:

```ada
package Item_Package is
type Item (UPC_In : integer) is tagged private;
-- Now all Items must have a UPC value upon declaration

function Create_New_Item(An_Item : in out Item; ...);
...

private

type Item is tagged
record -- Below, use discriminant to initialize components
UPC : integer := UPC_In;
OnShelf, InStock, ROLimit : integer;
RCost, WCost : float;
Name : Name_Type; -- Name is another structure
Loc : Location_Type; -- Loc is another structure
end record;
end Item_Package;
```

In the example, `UPC_In` for the tagged type `Item` must be defined whenever a variable is declared, as shown in:

```ada
nc : NameCost;
item1 : Item(32); -- This value initialized UPC for item1

Create_New_Item(item1, ...);
nc := Get_Item_NameCost(item1);
```

Notice that in this example, when declaring the variable `item1`, a `UPC` value must be supplied.

### 4.2.3 Support for Hiding

Hiding in *Ada 95* is supported by the physical location of primitive operations and the tagged type. Each package specification contains a publicly visible interface section, and a section hidden from the public view, as shown in the previous example for `Item_Package`. The hidden section is explicitly labeled private, with the public section by default being any section outside the private section. Primitive operations and any type or variable declarations which appear outside this private section are visible outside the package specification. Another mechanism for hiding is to place primitive operations, type, or variable declarations wholly within the corresponding package body. Each portion of a package housing a tagged type can thus contain components and/or primitive operations. However, it is typical that the private portion contains components, with the public portion composed of the primitive operations that manipulate the components. The public portion is that part of a package that is available to other packages (and users). The private portion of a class can only be accessed by going through a primitive operation provided for by the designer. For example access to the UPC component of `Item` is gained through the `Create_New_Item(An_Item: in out Item; UPC: integer, ...);` primitive operation. If the designer wished to provide an ability to update an `Item`’s UPC component, a `Change_UPC(An_Item: in out Item; New.UPC: integer)` primitive operation would be provided. Thus, the following code:

```ada
with Item_Package; -- Note: We will use the Item_Package without
use Item_Package; -- discriminants in the remainder of the examples

procedure main is
```

32
would result in the generation of a syntax error, since there is an attempt to print out the UPC of \texttt{item\_1}. Therefore, the public interface can be characterized as the union of all possible primitive operations for all potential users. A primitive operation intended for one or two specific users is available to all users, once it has been placed in the public area of a package specification.

4.2.4 Class-Wide Components

In \textit{Ada 95}, data can be shared by all instances of a tagged type through class-wide components. A class-wide component is any variable declared in a package enclosing a tagged type, but specifically outside of the tagged type declaration. Class-wide components allow software engineers to define variables that are shared by all instances, as opposed to instance variables (components specific to each instance). For example, consider a tagged type \texttt{Sales\_Info} for HTSS:

\begin{verbatim}
-- Package specification in one file
package Sales_Info_Package is
    type Sales_Info is tagged private;
    procedure Increment_Total(Sale: Sales_Info);
    procedure Print_Total_Orders(Sale: Sales_Info);
private
    Total_Orders : Natural := 0; -- Class-wide component
    type Sales_Info is null record;
end Sales_Info_Package;
-- Package implementation in another file
package body Sales_Info_Package is
    procedure Increment_Total(Sale: Sales_Info) is
        begin
            Total_Orders := Total_Orders + 1;
        end;
    procedure Print_Total_Orders(Sale: Sales_Info) is
        ... end;
end Sales_Info_Package;
-- The following would be in a separate file
with Sales_Info_Package; use Sales_Info_Package;
procedure main is
    Sale1, Sale2 : Sales_Info;
begin
    -- both increment Total_Orders!!!
    Increment_Total(Sale1);
    Print_Total_Orders(Sale2);
end main;
\end{verbatim}
In this example, the variable Total_Orders is defined to be a class-wide component and initialized to zero. When sale_1 and sale_2 are declared in the main program, despite the fact that they represent different instances of Sales_Info, they share the same storage for the Total_Orders variable. Thus, when sale_1 increments the total, then sale_2 will print the incremented value.

4.2.5 Forms of Inheritance in Ada 95

Inheritance, as a concept, allows information and behavior to be passed from one tagged type (parent type or superclass) to another tagged type (derived type or subclass) via an extension to the Ada 83 concept of derived types. In the HTSS, inheritance is very useful, since commonalities can be identified and grouped. For example, Deli_Item can be a subclass of Item, since it needs the many features of Item, requires additional private data to keep track of the cost per pound and the increment of sale. Other subclasses of Item for produce, baked goods, meats, etc., would also be appropriate.

Ada 95 supports two kinds of inheritance, differentiated by the portions of the superclass that are visible to descendants. The differences are traced to the where/how the derived type is physically declared. A derived type can be declared either in the same physical package of the parent, in a separate but otherwise ordinary package, or in a “child package”. The choice of how the child type is declared determines whether the inheritance is public or private.

Public Inheritance

Public inheritance is attained by placing the derived type is a separate “ordinary” package, giving the derived type access to any of the primitive operations or components of the parent type that are visible outside the package. These publicly visible primitive operations and components are those declared within the package specification of the parent, but outside of the private portion of the parent’s package. For example, consider the following public inheritance between Item and Deli_Item:

```ada
-- This package specification contained in one file
package Item_Package is
  type Item is tagged private;
  function Create_New_Item(An_Item: in out Item; UPC : integer, ...);
  ... -- The Rest of the Item_Package Specification As Before
private
  type Item is tagged record
    UPC, OnShelf, InStock, ROLimit : integer;
    RCost, WCost : float;
    Name : Name_Type;
    Loc : Location_Type;
  end record;
end Item_Package;

-- This package specification contained in another file
package Deli_Item_Package is
  type Deli_Item is new Item with private;
  -- "is new" means Deli_Item is derived from Item
  -- "with private" means that the details of the extension
```
to Item are hidden in the package’s private part

```plaintext
function Profit_Per_Lb(A_Deli_Item : Deli_Item) return float;
```

```plaintext
private
type Deli_Item is new Item with
  record
    Weight, Cost_Lb, Increm : Float;
  end record;
end Deli_Item_Package;

package body Deli_Item_Package is
  function Profit_Per_Lb(A_Deli_Item : Deli_Item) is
    begin
      return A_Deli_Item.RCost - A_Deli_Item.WCost;
      -- Above is an illegal reference to RCost & WCost
    end Profit_Per_Lb;
end Deli_Item_Package;
```

In the above example, the derived type `Deli_Item` inherits only primitive operations of the parent type, i.e., those operations that have an object of type `Item` in their profile. Only components of `Item` that are declared outside of the private section of `Item_Package` (none in this example) would likewise be inherited by `Deli_Item`. The private portion of `Item` is unavailable both to users of `Deli_Item` and `Deli_Item` itself. The above reference to `RCost` and `WCost` is therefore illegal since `RCost` and `WCost` are declared in the private section of `Item_Package` and are not inherited by types derived from `Item`. The derived type `Deli_Item` can have additional components, such as `Weight`, `Cost_Lb`, and `Increm`. However, none of the components added to the derived type may have the same names as those in the parent type.

Primitive operations, with or without the same names as the primitive operations in the parent type, may be added to the derived type. If a primitive operation with the same name is added to the derived type, it “overrides” the primitive operation in the parent type. Issues related to overriding primitive operations will be explored in Section 2.6. Note that if the parent type is declared private then the derived type must also be declared private.

**Private Inheritance**

Private inheritance is attained by one of two methods. A derived type may inherit privately from the parent by either placing the derived type directly in the same package of the parent type, or by placing the derived type in a “child package”. The latter is the preferred method, since it adheres more closely to the concept of abstraction. A derived type declared in a child package has access to both the public and private portions of the parent type’s package (but not the parent’s package body). For example, private inheritance with the derived type declared in a child package for HTSS would be as follows:

```plaintext
-- Package specification in one file

package Item_Package is
type Item is tagged private;

function Create_New_Item(An_Item: in out Item; UPC : integer; ...);
... As Before

private
type Item is tagged
```
record
    UPC, OnShelf, InStock, ROLimit : integer;
    RCost, WCost : float;
    Name : Name_Type;
    Loc : Location_Type;
end record;
end Item_Package;

-- Now, declare Deli_Item in a child package of Item (another file)
package Item_Package.Deli is
    type Deli_Item is new Item with private;
    function Profit_Per_Lb(A_Deli_Item: Deli_Item) return float;
private
    type Deli_Item is new Item with
        record
            Weight, Cost_Lb, Increm : Float;
        end record;
end Item_Package.Deli;
package body Item_Package.Deli is
    function Profit_Per_Lb(A_Deli_Item : Deli_Item) is
    begin
        return A_Deli_Item.RCost - A_Deli_Item.WCost;
        -- Above is now a legal reference to RCost/WCost
    end Profit_Per_Lb;
end Item_Package.Deli;

Declarations in the private part of Item_Package are known both in the private part and in the package body of Item_Package.Deli. The reference to RCost and WCost is legal since these components are inherited from Item by Deli_Item. Note that Deli_Item must be declared private since its parent type is declared private, yielding an effect similar to protected members in C++. Any primitive operations or components that are declared in the package body of the parent type are not available to derived types, giving an effect similar to private members in C++. A ‘practice’ of software engineering for object-oriented development is the use of only the public and inheritable (via child-package derived type declarations) portions of a type derivation. This allows information to be accessible to objects declared within the same inheritance hierarchy, without making the information visible to the outside world.

The example below shows the operation Create_New_Item (inherited from Item_Package) being directly used by the derived type.

    Peanut_Butter : Item;
    Hoagie : Deli_Item;

    Create_New_Item(Peanut_Butter, ...);
    Create_New_Item(Hoagie, ...);

Components belonging to the derived type Deli_Item either through inheritance (e.g., RCost) or through expansion (e.g., Weight) are not directly available to users of the object Hoagie.

4.2.6 Overriding Inherited Operations

A primitive operation declared in a derived type with the same name as a visible primitive operation in the parent type overrides the inherited primitive operation. Assume that Deli_Item needs a
different response than the `Time_To_Reorder` operation inherited from `Item`. Consider the following:

```plaintext
-- The parent package

package Item_Package is
    type Item is tagged private;

    function Create_New_Item(An_Item: in out Item; UPC: integer, ...);
    function Time_To_Reorder(An_Item: in out Item) return boolean;

private
    type Item is tagged record
        ... As Before
    end record;
end Item_Package;

-- Deli_Item in a child package overrides Time_To_Reorder

package Item_Package.Deli is
    type Deli_Item is new Item with private;

    function Profit_Per_Lb(A_Deli_Item : Deli_Item) return float;
    function Time_To_Reorder(An_Item : in out Deli_Item) return boolean;

private
    type Deli_Item is new Item with record
        Weight, Cost_Lb, Increm : Float;
    end record;
end Item_Package.Deli;
```

Using the previous package specifications, the following code can be developed:

```plaintext
Peanut_Butter : Item;
Hoagie : Deli_Item;
Need_Reorder : Boolean;
Create_New_Item(Peanut_Butter, ...);
Create_New_Item(Hoagie, ...); -- Operation inherited from Item
Need_Reorder := Time_To_Reorder(Peanut_Butter);
Need_Reorder := Time_To_Reorder(Hoagie); -- Overridden operation from Item
```

When ‘`Time_To_Reorder (Peanut_Butter)`’ is invoked, the version declared in `Item_Package` is called. For ‘`Time_To_Reorder(Hoagie)`’, the version declared in `Item_Package.Deli` is called. The ability to override primitive operations will play an important role in run-time dispatching.

### 4.2.7 Single vs. Multiple Inheritance

The previous discussion of inheritance focused on single inheritance, where any derived type has exactly one parent. Multiple inheritance expands this concept by allowing a type to inherit from more than one parent, when necessary. Multiple inheritance is acknowledging the simple fact that the world and its information does not occur in nice, easy to structure, hierarchies that are strict trees. Rather, there are many situations where the informational and/or operational requirements and characteristics of a class require it to inherit from more than one parent, thereby forming an
inheritance graph or lattice. Thus, the major benefit of multiple inheritance is a more accurate conceptualization of the application, since these special and unique situations can be modeled.
In HTSS, multiple inheritance can be used to more effectively indicate the relationships among different Items. For instance, consider the inheritance hierarchy:

```
Item
 |   |
+-------------------+------------+-----------+
ProduceItem | MeatItem   | BakeryItem | PharmItem
 |            |            |
+---------+---------+           |
|        |        | DeliItem |
```

In this situation, the common parent Item has two purposes. First, it allows us to abstract out all data/member functions that are common to items, regardless of their types. Second, when necessary, we can treat all of the instances, regardless of their actual type, as Items. Multiple inheritance can thus provide us with powerful information structuring and programming capabilities.

However, multiple inheritance does cause many different problems, at both conceptual and implementation perspectives. First, there is the possibility of naming conflicts, both for data portions and member functions, since a class may inherit from separate paths in the hierarchy. For example, in HTSS, ProduceItem and MeatItem might both have data that tracks when to remove the items from the shelve. However, for ProduceItem these values might be up to one week, while for MeatItem, days are more typical. Second, a common node might be inherited more than once, which is true in the example since DeliItem can acquire Item characteristics either from ProduceItem or MeatItem. This information must only be inherited once by DeliItem, which is further complicated by the fact that ProduceItem or MeatItem must share the same and correct Item instance that comprises the core of DeliItem. Third, naming conflicts and the inheritance of a common node more than once causes practical problems associated with a constructor, which must be more complex to handle these situations. This results in a runtime environment that is more difficult to manage and control (from the perspective of the compiler), and understand (by software engineers).

Putting these problems aside, in Ada 95 the spirit of multiple inheritance (i.e., viewing an object as any one of its parents) can be attained, even though multiple inheritance is not explicitly supported by language constructs. Multiple inheritance can be used when the derived type is truly the derivative of more than one parent, and clients of the type want to be able to view the derived type as any one of its parents. This is accomplished in Ada 95 using access discriminants, effectively parameterizing one record with another. For example:

```ada
type Outer is limited private;
private
  type Inner(Ptr : access Outer) is new (First Supertype)
type Outer is limited
  record
    ... (Second Supertype)
    Component : Inner(Outer’Access);
    ...
  end record;
```
Pictorially, this self-referential structure (taken from *Ada 95 Reference Manual*) has the following form:

```
+--------------------------+
|                         |
|                         |
| +-----------------+     |
| }                    |
| }                    |
| }                    |
| +-----------------+     |
| |                  |
| |                  |
| |                  |
| +-----------------+     |
| |                  |
| |                  |
| |                  |
| +-----------------+     |
| |                  |
| |                  |
| |                  |
| +-----------------+     }
```

The types `Inner` and `Outer` can both be extensions of other types. We can now declare an object of type `Outer`, thus creating the self-referential structure depicted above. For example, the type `Inner` could be an extension of another type `Linked_List_Node`, which has properties that can be used to construct binary trees. For additional details on achieving the effects of multiple inheritance in *Ada 95*, see *Ada 95 Rationale*, *Ada 95 Mapping/Revision Team*, *Intermetrics*, 1993.

### 4.2.8 Class-Wide Programming

Dispatching is one of the most important features that is offered by the object-oriented approach, and is defined as the runtime or dynamic choice of the primitive operation that is to be called based on the class type of the invoking instance. The concept is particularly useful in the presence of inheritance, since only in these situations can an instance be interpreted in many different ways. For example, the instance `Hoagie`, an instance of `Deli_Item`, can be also viewed as an instance of `Item`. Interpreting a group of instances as sharing the type of a common parent is important, since it allows us to cross class boundaries to aggregate different instances when necessary. This is analogous to the ability to construct a list where the different list entries have different types. Such a scenario would be needed for HTSS, if the entire inventory was to be scanned, rather than just a search for one item of a particular type. Consequently, dispatching offers many different benefits to software engineering, all related to extensibility and productivity, including: more versatility in the development of inheritance hierarchies and class libraries; the promotion of software reuse and evolution; and, the ability to more easily and effectively develop and debug generic code.

To support dispatching in *Ada 95*, a number of concepts are needed: the ability to support inheritance via derived types; the overloading of primitive operations in the derived class; the use of a hidden `tag` and the concept of a class-wide type. To serve as a basis for the discussion, consider the following package declarations:

```ada
-- Package specification for Item_Package

package Item_Package is
    type Item is private;
    ... As Before
```
-- now a primitive operation (possibly overridden in derived types)

procedure Print_Item_Info(An_Item : Item) is
begin
    Name_IO.put(An_Item.Name);
end Print_Item_Info;

private
... As Before
end Item_Package;

-- Package specification for Deli_Item in a child package

package Item_Package.Deli is
    type Deli_Item is new Item with private;
    ... As Before

procedure Print_Item_Info(A_Deli_Item: Deli_Item) is
begin
    Print_Item_Info(Item(A_Deli_Item)); -- Invokes the parent’s
    put(The_Deli_Item.Weight); -- operation and then
    put(The_Deli_Item.Cost_Lb); -- specializes
end Print_Item_Info;

private
... As Before
end Item_Package.Deli;

The Print_Item_Info procedure in Item_Package.Deli overrides the operation with the same name in the parent type's package (Item_Package). Since inheritance sometimes mimics specialization (the child type is exactly like the parent type, except...) designers may still want to invoke the parent types primitive operation as part of the overriding operation in the child. In the above example, Item_Package.Deli uses the parents Print_Item_Info by typecasting A_Deli_Item to Item, and then invoking Print_Item_Info on the typecast. This results in the parent’s Print_Item_Info procedure being invoked first, with the additional Deli_Item specific actions taking place next.

We will now add two more child packages for Produce and Bakery items as derived types, to facilitate the discussion of dispatching in the following section.

-- Package specification for Produce items

package Item_Package.Produce is
    type Produce_Item is new Item with private;
    ... Additional Produce Item Ops

procedure Print_Item_Info(A_Produce_Item : Produce_Item) is
begin
    Print_Item_Info(Item(A_Produce_Item)); -- Invokes the
    put(A_Produce_Item.Weight); -- parent’s operation
    put(A_Produce_Item.Cost_Lb);
end Print_Item_Info;

private
...Additional Produce Item Components
end Item_Package.Produce;
package Item_Package.Bakery is
    type Bakery_Item is new Item with private;

    -- Does Not Have Print_Item_Info Overriding Procedure
private
...
end Item_Package.Bakery;

Given the set of previous package definitions, the following main program can be developed:

procedure main is
    D1 : Deli_Item;
P1 : Produce_Item;
B1 : Bakery_Item
begin
    Create_New_Item(D1,111, PotatoSalad, ...);
    Create_New_Item(P1,222, Bologna, ...);
    Create_New_Item(B1,333, Celery, ...);

    Print_Item_Info(D1);
    Print_Item_Info(P1);
    Print_Item_Info(B1);
end main;

In the example, the primitive operation Print_Item_Info is defined in the parent type’s package, and may subsequently be overridden defined in derived type declarations. In the example, for the D1 and P1 instances, Print_Item_Info from the respective derived type declaration is called. For B1, the inherited version in Item is invoked.

Dispatching and Dynamic Binding in Ada 95

In Ada 95, dynamic binding (the run-time determination of which primitive operation is being invoked) is based on the hidden tag which exists for each expandable (tagged) type. This hidden tag (directly referable to as TAG) is based upon each types ‘family tree’. The family tree for a type is made up of all the types that have a common ancestor, including the ancestor itself. Each type in a type family has a unique tag. In the above example, Item’s family tree consists of Deli_Item, Produce_Item, and Bakery_Item along with the root Item. Each type family (T) has a polymorphic type designated as T’class. Semantically, T’class means that any of the specific types which comprise T’s type family will match (at run-time) with the type T’class. For example, if we have a formal parameter Deli_Order whose type has been stated to be Item’Class, then at run-time the variable expected to be of type Deli_Order could be passed an object of any of the types comprising Items type family. Consider the example given below:

type Item_Ptr_Type is access all Item’Class; -- General access type

Item_Ptr  : Item_Ptr_Type;
Item      : aliased Item; -- Can be pointed to by a general access type
Deli      : aliased Deli_Item;
Produce   : aliased Produce_Item;
Bakery    : aliased Bakery_Item;

42
In the above example, the access all Item'Class denotes a general access type. A general access type designates a pointer that may hold the address of any object within the type family (in this case Item). The command aliased allows an object to be pointed to (accessed) by a general Access Type. To get the access type Item_Ptr to point to a particular object, the attribute Access is used. The binding of the call to Print_Item_Info in the example is dynamic. It is impossible at compile time (for the compiler) to determine which Print_Item_Info procedure is being called. However, at run time, there is a guarantee (by the compiler) that a 'Print_Item_Info' will be found.

Polymorphic Formal Parameters

Polymorphic types can also be used when parameters to primitive operations are declared. In the following, a polymorphic parameter of type Item'Class can be used to print information for all the types in the type family based on Item.

```ada
procedure Put(Object_From_Item_Family : Item'Class) is
  begin
    Put_Line("Information for an Item");
    Print_Item_Info(Object_From_Item_Family);
  end Put; -- Dynamic binding of the above procedure call
```

The Object_From_Item_Family formal parameter can be matched at runtime with any object from the Item family tree.

4.2.9 Heterogeneous Collections of Objects

Dispatching, dynamic binding, and polymorphic formal parameters, when combined, can offer a powerful design and programming capability in Ada 95. To augment the package definitions for Items in HTSS, a generic linked list package is presented below. Each element of the list is implemented as a pointer to either a Node, or a type derived from Node. The linked list expects each type derived from Node to have a response to the message Print_All_Items.

```ada
package List_Package is
  type Node is tagged private;
  type Link is access Node'Class;
  procedure Insert_A_Node(A_Node: access Node'Class; List : in out Link);
  function Next_Node(A_Node: access Node'Class) return Link;
  procedure Print_All_Items(A_Node: access Node'Class);

  private
    type Node is tagged
      record
        Next : Link;
      end record;
  end List_Package;

package body List_Package is
```

43
procedure Print_All_Items(A_Node: access Node'Class) is
begin
Item_Ptr := Item_Ptr_Type(Store_Items);
while Item_Ptr /= null loop
Print_Item_Info(Item_Ptr.all);
Item_Ptr := Item_Ptr_Type(Next_Node(Item_Ptr));
end loop;
end Print_All_Items;
end List_Package;

We alter Item Package as previously defined so that Item is now derived from Node. All objects of Item'Class will then have the property that they can be elements in the linked list that Item is derived from. This is illustrated as follows:

package Item_Package is
  type Item is new Node with private;
  ... As Before
end Item_Package;

Given the aforementioned scenario, it is now possible to create a linked list to hold objects that are part of Item's type family. This is shown using a main procedure.

procedure main is
  type Item_Ptr_Type is access all Item'Class;
  Item_Ptr : Item_Ptr_Type;
  Store_Items : Link;
  Deli : aliased Deli_Item;
  Produce : aliased Produce_Item;
  Bakery : aliased Bakery_Item;
  begin
    ... Load Deli, Produce and Bakery with values ...
    Item_Ptr := Deli'Access;
    Insert_A_Node(Item_Ptr, Store_Items);
    Item_Ptr := Produce'Access;
    Insert_A_Node(Item_Ptr, Store_Items);
    Item_Ptr := Bakery'Access;
    Insert_A_Node(Item_Ptr, Store_Items);
    Print_All_Items(Store_Items);
  end main;

Dispatching occurs on calls to Insert_A_Node and Print_Item_Info. To understand this example in practice, consider a pictorial representation of the list of Items as referenced by Item_Ptr:

```
+----------+ +----------+ +----------+ +----------+
| Item | | Item | | Item | | Item |
| (Really| (Really| (Really| (Really|
| Potato | Bologna) | Celery) |
| Salad) |
+----------+ +----------+ +----------+
```
When `Print All Items` is called, the collection is traversed. The call to `Print Item Info()` would call the primitive operations in `Salad Item`, `Deli Item`, and `Produce Item`, respectively. Each of the specific `Print Item Info` functions will also likely call appropriate prints in any ancestor classes. With dispatching, it is possible to treat a set of instances as the same, if they share a common parent, when it's convenient and suits the needs of the application. Despite this common grouping, the underlying type of the instances can be automatically used at runtime to realize a specific behavior.

The major benefit and real power of dispatching relates to extensibility and software evolution. Suppose for HTSS that a new derived type, `Pharmacy Item`, was defined, with a similar structure to the other types derived from `Item`. To support this change, consider the revised `main` program that was previously given:

```ada
procedure main is
    type Item_Ptr_Type is access all Item'Class;
    Item_Ptr : Item_Ptr_Type;
    Store_Items : Link;
    Deli : aliased Deli_Item;
    Produce : aliased Produce_Item;
    Bakery : aliased Bakery_Item;
    Pharmacy : aliased Pharmacy_Item;
    begin
        ... Load Deli, Produce and Bakery with values ...
        Item_Ptr := Deli'Access;
        Insert_A_Node(Item_Ptr, Store_Items);
        Item_Ptr := Produce'Access;
        Insert_A_Node(Item_Ptr, Store_Items);
        Item_Ptr := Bakery'Access;
        Insert_A_Node(Item_Ptr, Store_Items);
        Item_Ptr := Pharmacy'Access;
        Insert_A_Node(Item_Ptr, Store_Items);

        Print_All_Items(Item_Ptr);
    end main;
```

While a new type derived from `Item` for `Pharmacy Item` must be defined, and a recompilation is necessary, there are no changes to the `Link` parent type. The `Print All Items` statement will work correctly, and consider `Pharmacy` as valid. Thus, the extension is seamless and easily done with a minimal impact on the existing implementation. Note also that sub-classes of `Salad Item` or `Meat Item` could be added in a similar fashion without the need to make significant changes.

### 4.2.10 Parameterized Types

The final, important concept that is needed for an object-oriented approach is a parameterized type, which supports genericity and promotes class reuse. A parameterized type is another term for a generic, and simply refers to a segment of code that can be applied to multiple data types, since it contains a parameter for type information. The prior example of a heterogeneous collection of objects required the container type to be developed (implemented as a linked list) before any children types of `Item`. Composition can also be utilized, and is more appropriate since the `Is A` relation does not logically hold between an `Item` and a `Node`. Through generics, an abstract list handler can be used as `Part Of` an `Item` list package, so that all inheritance conforms to the `Is A` test. The generic list handler can be written as an separate, independent entity. Consider the following generic `Ada 95` package for `List` that replaces the `List Package` from Section 2.9:
generic
type T is private;

package Generic_List_Package is
  type List_Type is limited private;
type Direction is (Start, End, Left, Right);

  function At_End (List : List_Type) return Boolean;
  procedure Traverse_List (To : Direction; List : in out List_Type);
  function Node_Value (List : List_Type) return T;
  procedure Insert_A_Node (A_Node : T; Position : Direction;
                           List : in out List_Type);

private
  type Node_Container;
type Link is access Node_Container;
type Node_Container is
    record
      Next, Previous : Link;
      Node : T
    end record;
type List_Type is
    record
      First, Current, Last: Link
    end record;
end Generic_List_Package;

The T referred to throughout the above package serves as a place holder for the type that will eventually be stored in the list, allowing the list to be developed while completely abstracting away the type of the list elements. The type T is referred to as a generic formal parameter, and must be provided at compile time. Upon compilation, the generic package is literally copied (with the type provided at compilation substituted for everywhere T appears in the generic package) and the resulting package is then compiled. For example, one has to simply declare a pointer to Item’s type family and instantiate the List package which then holds general access pointers to Item:

  type Item_Ptr is access all Item'Class;

package Item_List_Package is new
  Generic_List_Package(Item_Ptr);

use Item_List_Package;
Store_Items : List_Type;

-- To Add Items:
Insert_A_Node(Deli'Access, Start, Store_Items);
Insert_A_Node(Bakery'Access, Right, Store_Items);

-- To Display All Store_Items in List:
Traverse_List(Start, Store_Items);
while not At_End(Store_Items) loop
  Print_Item_Info(Node_Value(Store_Items).all);
  Traverse_List(Right, Store_Items);
end loop;
The interesting portion of this example is found in the use of `Generic_List_Package`, which is a generic declaration for a linked list. `Generic_List_Package` contains functionality for traversing, adding an element, etc. `Generic_List_Package` could have been developed for another application, and as such, was likely tested extensively. This means that there is a very high probability that there are neither syntax nor semantic errors in the implementation for `Generic_List_Package`. This in turn means that designing, implementing, and testing the HTSS will be facilitated by the strong likelihood that `Generic_List_Package` is error free. This is the real strength of generics; the facilitation of reuse that naturally leads to a reduction in design and development.

### 4.2.11 Sample Ada 95 Source Code

The following trivial Ada 95 source code files are provided so that the interested reader can better understand the inter-relationships between Ada 95 package specifications, package bodies, child packages. This also provides a complete example of how the object-oriented concepts that have been presented can work together.

---

```
-- filename : item_package.ads

package Item_Package is
  type Item is tagged private; -- an extendible type

  -- primitive operations, visible outside the package
  procedure Show_UPC(An_Item : in Item);
  procedure Set_UPC (An_Item : in out Item; UPC_in : in integer);

private
  type Item is tagged -- now complete the type declaration
    record
      UPC : integer; -- A component, hidden to all except
      end record; -- derived types in child packages
  end Item_Package;

-- filename : item_package.adb

with text_io; -- needed for using the generic integer_io package

package body Item_Package is

  package UPC_io is new text_io.integer_io(integer);

  procedure Show_UPC(An_Item : in Item) is
    begin
      UPC_io.put(An_Item.UPC);
    end Show_UPC;

  procedure Set_UPC(An_Item : in out Item; UPC_in : in integer) is
    begin
      An_Item.UPC := UPC_in;
    end Set_UPC;

end Item_Package;

-- filename : item_package-deli.ads
```

---
package Item_Package.Deli is
  type Deli_Item is new Item with private; -- a derived type

  -- primitive operations, visible outside the package
  procedure Show_Weight(A_Deli_Item : in Deli_Item);
  procedure Set_Weight (A_Deli_Item : in out Deli_Item; Weight_in : in integer);

private
  type Deli_Item is new Item with
    record
      Weight : integer; -- A component, hidden to all except
    end record; -- derived types in child packages
end Item_Package.Deli;

-- filename : item_package-deli.adb
-- --------------------------------------------------------------------
with text_io;
package body Item_Package.Deli is
  package weight_io is new text_io.integer_io(integer);

  procedure Show_Weight(A_Deli_Item : in Deli_Item) is
  begin
    weight_io.put(A_Deli_Item.Weight);
  end Show_Weight;

  procedure Set_Weight (A_Deli_Item : in out Deli_Item; Weight_in : in integer) is
  begin
    A_Deli_Item.Weight := Weight_in;
  end Set_Weight;
end Item_Package.Deli;

-- filename : main.adb
-- --------------------------------------------------------------------
with Item_Package; -- allows declarations declared in the
  -- package to be used in this procedure
with Item_Package.Deli;

procedure main is
  type item_ptr_type is access all Item_Package.Item'class;
    -- A type definition for a general access type
  item_ptr : item_ptr_type;
  item1    : aliased Item_Package.Item;
  deli1    : aliased Item_Package.Deli.Deli_Item;
    -- "aliased" indicates that this object may
    -- be pointed to by a General Access Type

begin
  Item_Package.Set_UPC(item1, 10);
  Item_Package.Show_UPC(item1);
  Item_Package.Deli.Set_UPC(deli1, 20); -- Inherited from parent
  Item_Package.Deli.Set_Weight(deli1, 83);
  Item_Package.Deli.Show_UPC(deli1); -- Inherited from parent

  48
Item_Package.Deli.Show_Weight(deli1);

item_ptr := item1'access;
Item_Package.Show_UPC(item_ptr.all);

item_ptr := deli1'access;
Item_Package.Show_UPC(item_ptr.all); -- legal, Deli inherits from item

-- Item_Package.Deli.Show_UPC(item_ptr.all); -- illegal

-- The Show_UPC defined (via inheritance) in the Deli child package
-- of item, expects a Deli_Type as a parameter, not the contents
-- of what a General Access Type is pointed to (even if the GAT is
-- pointing to a Deli_Type object)

-- Item_Package.Show_Weight(item_ptr.all); -- illegal

-- The above line is illegal since Show_Weight is not defined for
-- the type Item_Package.Item

-- Item_Package.Deli.Show_Weight(item_ptr.all); -- illegal

-- The above line is illegal since the Show_Weight is expecting
-- a Deli_Type, not the contents of what a general access
-- type is referencing

-- Note : If Item_Package and Item_Package.Deli had been named in a
-- "use" clause in addition to the two "with" clauses above, then
-- these two packages would be directly visible, eliminating the need to
-- specify which package a data type or operation comes from.

end main;

5 Object-Oriented Database Concepts

To provide a basis for discussions in Sections 6 and 7 of this chapter, object-oriented database concepts are reviewed. Two database systems have been chosen, Ontos [24] and ODE [2], for a number of reasons. First, Ontos represents the progress and approach of the commercial marketplace, while ODE is a research system that has been under development by AT & T. Second, and most importantly, each has a significantly different approach to providing database support via C++. In the case of Ontos, a programming-approach was chosen, where a class library has been developed for representing the persistent needs of object-oriented applications and systems. User and designer classes are defined as sub-classes of various Ontos library classes that supply all relevant 'implementation' for persistence. On the other hand, ODE has taken a database approach, where C++ has been extended to O++ to include database capabilities such as integrity and triggers. This extensions can then be preprocessed out in a transparent fashion from the end-users. Other extensions allow persistent variables to be declared. Thus, together Ontos and ODE capture the two classic approaches to supporting persistence for the object-oriented paradigm.

Ontos, as shown in Figure 2.7, is a true DBS that supports the client and server architecture, allowing multiple servers, multiple DBs per server, and concurrent access across the entire system by multiple users [24]. To confer persistence to a C++ application, software engineers must derive their classes from a special Ontos class library. Through the library, the engineer is protected from
knowing some of the actual details related to accessing database objects. So, conceptually, a check-in and check-out process is used to activate (deactivate) objects from (to) DB. Other system features and capabilities include: a single name space, hierarchically organized, for each DB; optimistic and pessimistic concurrency control to suit different application needs; and the storage of class declarations (e.g., the schema) within the DB.

To support persistence, an extensive class library is employed, as shown in Figure 2.8. A C++ class can acquire persistence if it inherits from the Ontos class Object. A set of primitive types is also supported for defining data members of C++ classes. In addition, since a common activity and major purpose of a DBS is to allow multiple, similar instances to be grouped under a single name, a number of Aggregate types have been provided. Set, List, and Association offer different semantics regarding the creation, insertion, and deletion of instances within an Aggregate. In addition, all of the instances of a class can be automatically and transparently saved in the database provided that Aggregate has been identified as an extension of the class. To do this, a special switch must be set when invoking the Ontos compiler to activate the extension feature.

To demonstrate the various Ontos concepts, consider the following definition of the Item class, which has acquired persistence by inheriting from Object of Figure 2.8:
#include <Object.h>

class Item : public Object {
    protected:
        int UPC, OnShelf, InStock;
        int ROLimit; // RO for Re-Order
        char* Name;
        float RCost, WCost; // R - Retail; W - Wholesale
        Loc Location; // Loc is another class

    public: // The Constructor - obj_name for Identity
        Item(char* obj_name=(char*)0);
        ~Item(); // The Destructors
        void Destroy(OC_Boolean aborted=FALSE);

        Item(APL* theAPL); // Activation Constructor
        Type* getDirectType();
While there are many similarities, there are also some noticeable differences. These occur as part of the constructor definition, and the need for an activation constructor and the Destroy and getDirectType operations. However, the majority of the differences are not in the header declaration, but are mainly in the implementation of the various member functions:

```c++
#include "Item.h"
extern Type* ItemType;

// The Constructors and Destructors
Item::Item(APL* theAPL) : (theAPL) {}
Item::Item(char* obj_name) : (obj_name) {
    // Bring Objects from Database
    // Not Explicitly Called
    initDirectType(getDirectType());
}
Item::~Item() { Destroy(FALSE); }
void Item::Destroy(OC_Boolean aborted)
    { // User Code to Release Resources
        if (aborted) Object::Destroy(aborted);
    }
Type* Item::getDirectType()
    { return ItemType; }
    // Ptr to Type for Each Class
    { return ItemType; }

// Remainder are for user defined member functions
```

In general, the changes have been localized in the member functions as illustrated above. Clearly, the software engineer must understand persistence and its impact on the development of the class library in order to effectively use Ontos. Derived classes are supported in a similar fashion; they simply inherit as described in Section 4.1.6 and require their own specialized constructors and destructors.

The storage and removal of instances to and from the disk are encapsulated in two special Ontos functions, putObject and deleteObject. Each of these must be defined for each class that requires persistence. For the Item class, we would have:

```c++
// Store an Item to Disk
Item::putObject(deallocate);

// Delete an Item From Disk
Item::deleteObject(deallocate);
```

The coding of these member functions requires a significant amount of knowledge of Ontos, since numerous database system calls are required. However, once an engineer has been able to confer persistence for one class, supporting persistence for other classes follows the same methodology. The key strength in this approach is that the database system code is, for the most part, encapsulated within the implementation of the various Ontos-required member functions. Finally, if a software engineer wants to group multiple, related instances together so that they can be accessed as a single unit, Ontos system-defined classes can be utilized. For instance, the Set type is an aggregate that provides all of the appropriate data and member functions to implement sets and their associated operations. This is demonstrated using HTSS with the ItemCollection example from Section 4.1.9:
#include <Set.h>
#include <Object.h>
extern Type* ItemType;

class ItemCollection : public Object {
protected:
    int Total_Items;
    Set ItemSet(ItemType, (char*)0);

public: // Constructors and Destructors
    void Print_All_Items();
    // Other Appropriate Member Functions
};

void ItemCollection::Print_All_Items() {
    SetIterator my_iterator(&ItemSet);
    while (my_iterator.moreData()) {
        Item* element = (Item *) (Entity *) my_iterator;
        element->Print_Item_Info();
    }
}

While this is similar to the earlier example, there are a number of differences that are Ontos-specific. Most notable is the iterator concept, where different iterators are provided by Ontos to allow groups of instances to be searched. The iterator hides the implementation in the sense that there are no direct pointer links that need to be accessed.

ODE, short for Object Database and Environment, has been under development at AT&T Bell Labs, Murray Hill [2]. ODE has a complementary approach to Ontos, by defining O++, which is a persistent version of C++ that has been augmented with some of the typical database features that are found on all database systems. This includes: constraints to control the values of data for supporting consistency, versions to allow multiple copies of the same data to be tracked over time, and triggers to support the automatic change of data both within and across classes. When examining ODE it is difficult to ascertain if O++ is a programming language, database language, or both. On the one hand, since O++ is an extension of C++, it would be easy to say that it’s a programming language. However, unlike the C++ approach where class libraries are a norm, in O++ persistence is supported by variable declaration, as shown in:

class Item {
    protected:
        int UPC, OnShelf, InStock;
        int ROLimit; // RO for Re-Order
        char* Name;
        float RCost, WCost; // R - Retail; W - Wholesale
        Loc Location; // Loc is another class

    public: Item(...);
            ~Item();
    // Remaining Functions As Before
};

persistent Item *I1; // Declare an Item
In the example, \( I_1 \) is defined as a pointer to a persistent \( \text{Item} \). Given this definition, whenever the constructor of \( \text{Item} \) is invoked, as shown with the \text{new} \ command, the database is automatically informed that the instance will be persistent. Thus, unlike Ontos, in ODE, the details regarding storing and retrieving the database object are more transparent to end users.

To support constraints and triggers, extensions to \text{C++} \ are provided that allow \text{O++} \ classes to be designed and specified in a consistent fashion. In the example below, notice that the definition of constraints and triggers occur after the protected and public portions of the class, and consequently, have no impact on these portions.

```cpp
class Item {
    protected:
        // As before
    public:
        // As before

    constraint:
        RCost >= WCost;
        InStock >= OnShelf;

    trigger:
        order(): InStock <= ROLimit ==> Place_Order(this, eoq);
};
```

The \text{constraint} \ portion of the \text{O++} \ class allows boolean associations among the data items to be specified. These constraints must be satisfied at all times. Similarly, the \text{trigger} \ portion of the class indicates that when the \text{order} \ member function is called, it could result in an automatic call to \text{Place_Order} \ if more stock is required. To utilize versions in an application, a call to \text{newversion(curr_object_id)} \ would create a new version, while a call to \text{previous(old_object_id)} \ asks to use an existing version.

6 Requirements for Object-Oriented Databases

Our discussion to this point has focused on an examination of object-oriented programming and databases, using actual systems as a means to convey the various concepts. As a means to organize the various ideas into a meaningful context, we review an excellent article by Zdonik and Maier on the nature and requirements for a database system to be considered object-oriented [38]. While their perspective has a definite database bias, they have incorporated many important programming issues in their work. In their efforts, a number of different models are proposed, that characterize the capabilities of a DBS, and OODBs, and finally, an OODB with many programming language features. In the process, they raise many questions that are at the crux of the difficulties in integrating the two approaches.

To begin, consider the essential and frequent features of a DBS, that can be used to characterize the minimal database requirements. The essential features are what one would typically expect from a DBS:

- **Model and Language for DBS**: The model provides the representational capabilities, while the language provides the concrete means to define and access databases.

- **Relationships for Associations Among Database Entities**: This feature acknowledges the fact that information can’t exist in isolation, but the ability to relate a piece of information to other information is critical.
• Permanence for Creating a Repository: Long-term storage is a necessary and indispensable feature for any database system.

• Sharing to Control Access and Consistency: This feature recognizes that multiple users require access to the same databases at the same times, and this access must be controlled and managed to insure consistency.

• Voluminous Storage for Ever-Increasing Database Size Needs: Despite increases in main memory and disk sizes over the past few years, there is always the need for storing larger and larger databases.

This set of essential features is augmented by a set of four frequent features:

• Integrity Constraints: An important component of information consistency as defined in Section 1.4 of Chapter 1.

• Authorization: Another dimension of information consistency for controlling access and visibility to the database.

• Querying: Provide some means to access and modify stored information.

• Separate Schema: The schema is often referred to as meta-information, since it contains the information about the different databases, e.g., their structural makeup, sizes, locations, etc.

However, as argued in Section 5 of Chapter 1, the first two frequent features are essential in our view, since both integrity constraints and authorization relate to the fourth essential feature, i.e., without them, sharing and consistent access are not easily attained.

The threshold model proposes a set of minimum requirements to be characterized as an object-oriented database system. Starting with the essential features (expanded to include integrity and authorization), the threshold model adds the following key characteristics, to add, in Zdonik and Maier’s view, the core object-oriented capabilities:

• Object Identity: This feature of the threshold model is necessary to allow each database object to be both identifiable and distinguishable.

• Encapsulation and Types: In addition to encapsulation, there must also be the ability to dynamically create new types via a system library or by primitives that allow user-defined types to be specified.

• Complex State: In some sense, complex state is a realization of the second essential feature, and requires that objects have the ability to refer to (or be referred by) other objects. This, in turn, promotes the ability to define and support arbitrarily complex objects.

There is much to support the idea that inheritance must also be included in the threshold model. This has been argued most strongly in Section 3.3 of Chapter 1, since inheritance is a major feature that distinguishes the object-oriented approach from ADTs. Without inheritance it appears that the threshold model supports an ADT approach for DBSs.

While the threshold model has given the minimum requirements for an OODBS, this view has been strongly influenced by database considerations. If we are to evolve to a common integrated, programming-language/database domain, there are key programming concepts that are necessary to provide the basis for this evolution. The first is dispatching, which is a critical object-oriented concept, as demonstrated in Sections 4.1.8 and 4.1.9. Next, there must be support for polymorphism,
so that type-independent classes can be defined. Polymorphism has strong ties to generics and promotes software reuse. Third, the need for typing, which is an important concern for programming languages. Type systems vary across different languages, and are often categorized via the degree of type checking (strong vs. weak) and the timing of type checking (dynamic or runtime vs. static or compile time). In practice, different combinations of these categories are supported by different programming and database languages. The last concept that must be addressed involves persistence, since to integrate programming and databases, there must be an understanding of how and when persistent instances are stored. Zdonik and Maier offer a number of possibilities, including: Upon Creation? When Connected to a Persistent Object? When Explicitly Placed in a Persistent Store? When an Explicit Message is Received? Upon Instantiation from Persistent Store? They do not attempt to choose the best approach; instead they argue and discuss the relative merits of each. An interesting issue that is left to the reader to consider is the approach that is taken by Ontos, ODE, or some other OODB.

Given the threshold model and its enhancement with dispatching, polymorphism, typing, and persistence, Zdonik and Maier arrive at a reference model that, in their view, encompasses the requirements of object-oriented programming and databases. The reference model includes: structured representations for objects; persistence by reachability; typing of objects and variables; three hierarchies for the specification of types, implementation of data/methods, and classification of objects; polymorphism; collections; name spaces; queries and indexes; relationships; and versions. Many of these features have their origins in PLS, roots in DBS, are useful in OS, are critical for software development, and involve information consistency. However, the more interesting features are those not on the list, namely, acknowledgement of software extensibility via evolution and reuse. Note that the specification, implementation, and classification hierarchies [38, pages 16-17] do support these concepts. But, it seems that since they are so critical, they should be explicitly mentioned as part of the reference model.

Finally, we would be remiss if we did not mention the work Zdonik and Maier have done related to subtyping and inheritance, which may be one of the most significant contributions to the field. When developing inheritance hierarchies, there are four considerations that dominate one’s decisions:

1. Substitutability: The ability to view instances differently at different times, i.e., if B is a subtype of A, then B can occur where A is expected. This was demonstrated in Section 4.1.9 via dispatching, virtual functions, and friends in C++.

2. Static-Type Checking: The mandate that at compile time, everything related to subtyping and usage of types and instances is known, i.e., there is no need for dynamic, runtime type checking.

3. Mutability: Allowing an instance to change its type during runtime. This change is not restricted to an ancestor or descendant (as discussed in Section 4.1.9), but may occur across an inheritance hierarchy.

4. Specialization via Constraints: This subtyping feature provides the ability for the further restrictions of instance characteristics in the presence of inheritance. For example, the Item class from HTSS might have a constraint $\text{RCost} \geq \text{WCost} + \text{Factor}$ that relates retail and wholesale costs. This constraint might be specialized in DeliItem to $\text{RCost} \geq \text{WCost} + 2 \times \text{Factor}$.

Given these four subtyping features, Zdonik and Maier correctly claim that only three of four can ever exist in a single model! If all four could exist, then it would represent an ideal model with powerful capabilities and versatility, at least with respect to subtyping. However, the premise is that
if you take any three, and establish a scenario for subtyping where they are meaningful, the attempt to add the fourth feature will result in a conflict and/or contradiction. This is left as an interesting and important exercise.

7 Object-Oriented Programming vs. Databases

In Section 1.3 of Chapter 1, when motivating the role of the database, a brief discussion on the commonalities and differences between a DBS and a PLS was presented, to understand and then bridge the impedance mismatch between the two approaches. This section continues this discussion with the goal to provide an in-depth investigation of the merging of these two approaches to object-oriented systems. Sections 3, 4, 5, and 6 of this chapter have clearly illustrated both the capabilities and differences that exist between an OOPLS and an OODBS. What has not been totally resolved is a determination of their ‘ideal’ combination. This section will propose such a solution. To set the context for that presentation, the similarities and differences between OOPLSs and OODBSs are considered in more detail, from a general perspective, and at a practical level by comparing C++ and the Opal data model/language [25] of the Gemstone OODBS [20]. Note also that the reader is referred to [13] for a more comprehensive treatment of this material.

Just as DBSs and PLSs have similarities and differences, as discussed in Section 1.3 of Chapter 1, so too do OODBSs and OOPLSs. When considering the latter object-oriented systems, we are moving closer and closer to tomorrow’s view, as was indicated in Figure 1.1. There still remain many philosophical differences. Most notable is the tradeoff between information hiding in an OOPLS and data security in an OODBS. Hiding has no concept of a user, and emphasizes a public interface that is available to all. On the other hand, security mandates a very controlled environment with respect to information, with an assumption of ‘private’ until authorized access is granted. Inheritance is also another major difference in the two approaches. In an OOPLS, inheritance is touted for uniformity, dispatching, reuse, and evolution. This contrasts with the database interpretation when work on generalization was first conducted, where the underlying motivation for inheritance was to allow “... relevant details to be introduced in a controlled manner ...” [31, page 105], which emphasizes abstraction. Once philosophy is put aside, implementation considerations must be addressed. This raises many important questions:

• Who Executes Method Code? OOPLS? OODBS?

• Are Code Optimization, Static Semantic Program Analyses, etc., Still the Responsibility of an OOPLS?

• Are Disk I/O, Concurrency Control, Recovery, etc., Still the Responsibility of OODBS?

• What are the Shared Features of the Combined System?

Many of these questions have been discussed in part, and collectively, are important to build a successful bridge between object-oriented programming and database approaches.

From a practical perspective, another logical step to understand the issues is to identify the features in an OOPLS, like C++, that would need to be upgraded or enhanced to include database concepts. For example, there are clearly ties between C++ and OODBS classes, as discussed in Section 5. In addition, the public, protected, and private components of a C++ class do have parallels to an OODBS which would want to initially make all components private (or protected), and migrate a subset over time to the public portion of a class. Friends in C++ are also related to security in an OODBS, since the friend capability is a controlled technique to grant access. There are also
tenuous ties between methods in C++ and transactions in an DBS, and typing in C++ and integrity constraints. But, clearly these associations are minimal, at best.

The complement to the above discussion is to examine an OODBS to determine the degree to which it coincides with an OOPLS. For this practical comparison, we choose the Opal data/model language of the Gemstone OODBS. Opal evolved from the Smalltalk OOPLS, and like Ontos, provides a class library through which user-defined Opal classes can acquire persistence. The Item and DeliItem Opal class for HTSS are:

```smalltalk
Object subclass: #Item
  instVarNames: #(UPC OnShelf InStock ROLimit Name RCost WCost Location)
  classVars: ()
  poolDictionaries: ()
  inDictionary: UserGlobals
  constraints: [(#UPC, Integer), (#OnShelf, SmallInteger), (#InStock, SmallInteger), (#ROLimit, SmallInteger), (#Name, String), (#RCost, Float), (#WCost, Float), (#Location, Loc)]
  isInvariant: false.

Item subclass: #DeliItem
  instVarNames: #(Weight CostLb Increm)
  classVars: ()
  poolDictionaries: ()
  inDictionary: UserGlobals
  constraints: [(#Weight, Float), (#CostLb, Float), (#Increm, Float)]
  isInvariant: false.
```

In Opal, the subclass statement is employed to inherit from the class library or a user-defined class. Instance variables are the attributes of a class. Class variables are shared by a class and all of its instances, which is equivalent to the static storage class in C++. Pool dictionaries allow classes to be shared both between users and/or schemas. The inDictionary portion of a class is used to identify the location where class is stored, to facilitate reuse. The constraint portion of the class, while intended for indexing, can also be used to specify types for instance variables, as the examples indicate. However, all variables can be untyped in Opal. Lastly, the isInvariant portion of a class sets a condition that indicates whether an instance is changeable after it has been created.

But, is Opal an OOPLS? Clearly, there are many correspondences between Opal and C++, including: instance variables in Opal and members in C++; similar treatment of inheritance; class variables in Opal to static in C++; and constraints in Opal, when used for typing, to typing in C++. Other associations between Opal and C++ are not as obvious. Are pool dictionaries equivalent to friends? Is inDictionary similar to include libraries in C++? What are the equivalences of constraints (for indexing) and isInvariant in C++? Thus, while there are some commonalities, there are some significant differences.

Perhaps the real problem with trying to turn C++ into an OODBS or Opal into an OOPLS is one of approach. In either case, we are attempting to retrofit and extend an existing system to a new one. Wouldn’t this be similar to trying to turn an apple into an orange? Instead, the focus must be on identifying a unified set of features that represent a combined OOPLS and OODBS approach, much like the reference model of Zdonik and Maier. Such a set of common and critical features is given below, augmented with a discussion of the features with respect to C++ and Opal:

**Typing:** Does the language require static or dynamic typing and if so why was that particular choice made?
Universal Class: Is there a class at the top of the hierarchy and what are the implications of its existence?

Opal is dynamically typed, from a common object class, thereby supporting a database perspective where there is a unique identifier for each instance. Messages (methods) in Opal are found at runtime by searching the class and all of its ancestors. Unlike Opal, C++ is statically typed, and does not support persistency. However, C++ does guarantee that a correctly compiled program will always find the member function (method) at runtime.

Representation: Does the language facilitate the user’s ability to represent the data to implement an ADT?

This is a very difficult issue in both Opal and C++. In Opal, complex data structures are difficult to represent, with explicit references between types not supported. While C++ has pointers, arrays, and so on, adequate support for genericity and multiple inheritance has not been proven.

Encapsulation: How is the representation (partially) hidden to insure that the state of an object is not corrupted?

Inheritance: How is inheritance handled and what benefits are derived (no pun intended) from the technique?

These two features have been heavily discussed for C++ and Opal. Note that there are strong ties to security, concurrency, and constraints, and that problems with both multiple inheritance and persistency must be reconciled.

Partition: Can data be partitioned for large projects?

This feature is clearly not supported by C++. Opal offers a limited degree of partitioning, since the different dictionaries can be used to store different projects.

Security: How are users and security handled?

Concurrency: How is data shared and access?

Constraints: Can data be constrained to meaningful values?

While security relates to both encapsulation and inheritance, the main difference is that security is a dynamic process. At the type-level, security privileges can be defined. But, these must be then enforced at runtime, in much the same fashion that substitutability and dispatching are enforced. Concurrency is related to security, and more importantly to information consistency. Constraints must transcend the typing available in programming languages so that both the correctness and relevance of information can be insured. Note that none of these features are supported by either C++ or Opal.

Mutability: How is (non) mutability of objects managed?

A difficult problem, for both OOPLS and OODBs. Much work on schema evolution for OODBs has been conducted [5]. Muting from an instance to its ancestors via inheritance is understood in OOPLS, but to descendants and across siblings causes problems.

Class Reuse: How are classes and their associated methods stored for promoting accessibility and reuse?
Physical Database: How is data stored (accessed) and what is the impact on recovery, versioning, reuse, etc.?

The class library concept, in both OOPLSs and OODBSs, has been well established. However, the degree to which each of these systems provide environments that promote and facilitate reuse has not been established. Nevertheless, the storage of class libraries will have to rely on the physical database.

While there are similarities between our list and the reference model, the main differences are traced to scope, since we have included the engineering and design process as an integral part of a unified OOPLS/OODBS. An interesting question that is left as an exercise is the degree to which Ontos and/or ODE support this categorization.

8 Looking Ahead and Further Readings

In this chapter, we have covered a broad spectrum of design approaches, ranging from traditional techniques through a combined OOPLS/OODBS approach. In the course of the discussion, we observed that both traditional and object-oriented techniques lack the necessary engineering rigor for design analyses and evaluation, i.e., it is hard to determine when a design is ‘complete’. This is a serious shortcoming that was motivated in Chapter 1. However, we believe that the approach which begins in Chapter 3 and continues through Part II of the textbook can offer the required methods, techniques, analyses, and engineering discipline for eventually addressing this issue. Through this chapter we also examined a wide variety of object-oriented approaches (i.e., C++, Ada95, Ontos C++, ODE and O++, and, Gemstone and Opal) that contained a diverse range of capabilities and functions. In addition, we noticed that there was often conflicting and contradictory terminology that must be reconciled to present a more unified view of object-oriented design and development.

The work that was presented in Sections 4, 5, 6, and 7, is strongly related to object-oriented design and analyses, from motivational and foundational perspectives. Section 4 not only highlighted the capabilities of C++ and Ada95, but also indicated the more advanced object-oriented concepts that are critical for successful design. In Section 5, ODE and Ontos provided analogous information for OODBSs. Section 6 in its review of the Zdonik and Maier [38] article, provided the higher ground by focusing on critical features for all OODBSs regardless of their actual implementation approaches. In later chapters the impact of this work on our own efforts will be apparent. Finally, Section 7, by comparing techniques to evolve an OOPLS (OODBS) to include OODBS (OOPLS) capabilities concluded that either approach is flawed. Instead, synergy can be achieved by proposing a unified approach, which is our intent via the object-oriented design and analyses model of Chapters 5 through 10.

Given the motivation and background from Chapters 1 and 2, we can begin the process of examining information engineering by considering the content, usage, and importance of a specification in Chapter 3. The major questions that must be answered to successfully develop a specification include:

- What does a specification contain?
- Who needs to use a specification and at what times?
- Does a specification for an object-oriented design/implementation need to be object-oriented?
- What are the features of a ‘good’ specification?
- How is the specification utilized during design, development, and maintenance stages?
While Chapter 3 explores some of these questions, later chapters will also involve the fifth question since it is critical for integrating the entire design and development process. Part II of the book, starting in Chapter 4, will explore, in detail, object-oriented design and development techniques.

Lastly, the list below represents a set of readings that is intended to either coincide or follow the material of this chapter, with the purpose of reinforcing the concepts. We consider both the Budd [10] and the collection of readings by Zdonik and Maier [37] to be companion books for a graduate course on the material presented herein. Thus, throughout the remaining chapters, both of these books will be heavily referred to in the related readings section.

- D. Rine and B. Bhargava, “Guest Editors’ Introduction: Object-Oriented Computing”, pgs. 6-10, in [17].
- L. Cardelli, Semantics of Multiple Inheritance”, pgs. 59-83, in [37].
- A. Snyder, “Encapsulation and Inheritance in Object-Oriented Programming Languages”, pgs. 84-91, in [37].

The first reading provides additional material on object-oriented programming concepts, uses examples in four languages, and is intended to supplement the material in Section 4.1. The next three readings should be done in parallel with Chapter 2, since they provide another perspective on the basic or core object-oriented concepts. Finally, the last four readings are intended to be assigned as Chapter 3 is being discussed in class. These readings reinforce and extend the concepts of Chapter 2.
Chapter 2 Exercises

1. Discuss and demonstrate other traditional design approaches for specifying the basic structure and functional characteristics of the high-tech supermarket system (HTSS). Some possible approaches include: finite state machines, petri nets, structured design, etc. (Note: See any introductory software engineering textbook [14, 28, 32] for other approaches.)

2. Explore more aspects of the HTSS by expansion, extension, and refinement of the data-flow diagrams given in Figures 2.1 and 2.2 of Section 2.

3. Examine additional database features and interdependencies of the HTSS by utilizing ER and extended-ER techniques for conceptual database design. Use Figure 2.3 of Section 2 as a starting point for this problem.

4. Semantic database models such as IFO [1], SDM [16], and PDM [21], also support conceptual database design using techniques that are similar to the ER approach. Choose one of these database models and examine its ability to support the HTSS application.

5. Which, if any, of the traditional design approaches (e.g., DFDs, ERs, Modules), support and/or promote information consistency as discussed in Chapter 1?

6. Implement a subset of the HTSS application using a language which supports module concepts. Candidate languages include Modula-2 and Ada83.

7. Write a paper that critiques the degree to which DFDs, ER Diagrams, and Programming with Modules, supports the claims of the object-oriented paradigm as discussed in Chapter 1 (i.e., stresses modularity, increases productivity, controls information consistency, promotes software reuse, and facilitates software evolution).

8. Expand and elaborate on the HTSS solution using ADTs and the notation given in Figures 2.5 and 2.6, and Section 3 of this chapter. Clearly identify any interdependencies that must exist between the different levels of ADTs that are used in your solution.

9. Develop portions of the HTSS using the object-oriented programming language of your choice (e.g., C++, Ada95, Objective-C, ObjectPascal, SmallTalk, CLOS, Modula-3, etc.).

10. Investigate the following concepts in C++:
   
   a. Overloading by developing a complex number class where all of the expression, comparison, and assignment operators are redefined to manipulate and return complex numbers.
   
   b. Inheritance, both public and private derived classes, to obtain a clear understanding of where and when protected data can be used and passed onto subclasses.

   c. Dispatching, virtual functions, and friends using the classic example of an employee database at a corporation or university. (Note: for this problem, you should develop an inheritance hierarchy design as a first step in developing your classes.)

   The majority of this exercise may be repeated for concepts that apply to Ada95, Modula-3, or other object-oriented languages.

11. Do the C++ concepts: class, static member, overloading, inheritance, virtual functions, and friends; support or inhibit the claims of the object-oriented paradigm as listed in Exercise 7. Use examples when necessary to bolster your arguments.
12. Write a detailed paper that compares an existing commercial or research object-oriented database system (e.g., Ontos, ODE, GemStone, Postgres, etc.) against the reference model proposed by Zdonik and Maier and reviewed in Section 6.

13. As part of our discussion in Section 6 on the Zdonik and Maier article, persistence is a critical concept that must be supported to achieve the combination of programming and database domains. In this discussion, a number of ways to attain persistence was proposed. Using that list as a starting point provide a detailed analysis of the problems associated with achieving persistence. Take a stand on what you consider to be the “best” solution.

14. Evaluate the reference model of Zdonik and Maier (see Section 6) with the unified features proposed in Section 7. How do the two lists compare? differ?

15. Write a detailed paper that compares an existing commercial or research object-oriented database system (e.g., Ontos, ODE, GemStone, Postgres, etc.) against the unified features proposed in Section 7.

16. Zdonik and Maier claim that the four features of subtyping (substitutability, static-type checking, mutability, and specialization via constraints) cannot all exist at one time in any one system. Rather, only 3 of the 4 features can ever be supported in a single system. Consider the 4 different combinations that group 3 of the 4 features. Do all 4 combinations make sense? If not, why not? If so, why does the addition of the 4th feature cause a problem? For each combination that does make sense, what are the underlying system benefits and advantages?

17. In Zdonik and Maier, a number of future research areas related to the object-oriented approach have been proposed:

<table>
<thead>
<tr>
<th>Query Languages</th>
<th>Version/Config. Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deductive/Logic DBS</td>
<td>Temporal/Time Issues</td>
</tr>
<tr>
<td>Optimization</td>
<td>Storage Management</td>
</tr>
<tr>
<td>Parallelism</td>
<td>Non-Standard Architectures</td>
</tr>
<tr>
<td>Tools and Methodologies</td>
<td>HyperText</td>
</tr>
</tbody>
</table>
| Cooperative Work | Others???

Choose and discuss three areas. What other areas have been omitted?

18. The issue of completeness was raised in Section 8, which is a fundamental problem regardless of the domain. In any type of software design and development activity, it is impossible to know whether the end product is complete. While this issue will be discussed in detail in later chapters, at this stage it is important to understand the role that object-oriented concepts and principles can play in the attainment of completeness. Thus, please discuss the following considerations in detail:

- The impact of encapsulation, hiding, and inheritance.
- Advanced features such as polymorphism, dispatching, and parameterized types.
- The reference model as presented in Section 6.
- Extensibility, reuse, and evolution.
- Substitutability, static-type checking, mutability, and specialization via constraints as presented in Section 6.

Your answers must address if and how each consideration impacts (either positively or negatively) on completeness.
References


[38] S. Zdonik and D. Maier, “Fundamentals of Object-Oriented Databases”, in [37].