1997

Object-Oriented Development in Creating Software Systems

Brooke Frost
College of Saint Benedict/Saint John's University

Object-oriented Development in Creating Software Systems

A THESIS

The Honors Program

College of St. Benedict/St. John's University

In Partial Fulfillment

of the Requirements for the Distinction

and the Degree Bachelor of Arts

In the Department of Computer Science

by

Brooke Frost

May, 1997

Approval Page

PROJECT TITLE: Object-oriented Development in Creating Software Systems

Approved by:

J Andrew Holey
Associate Professor of Computer Science

Lynn Ziegler
Associate Professor of Computer Science

Paul Pladson
Associate Professor of Accounting

Noreen Herzfeld
Chair, Department of Computer Science

Director, Honors Thesis Program

Director, Honors Program

Table of Contents

Introduction

Since the late 1980s, object-oriented development has become quite popular and well-known to software programmers (Sommerville 248). This popularity can be attributed to the hope that this methodology will become the key to increased productivity, improved reliability, and fewer problems in computer software development (Budd 1). There are three components that are a part of object-oriented development. One component is object-oriented analysis which involves the creation of an object-oriented model based on the application domain of the software system. Programmers then must define the requirements that outline what is needed to build the software product. The second component is the object-oriented design which is when programmers develop an object-oriented model based on the defined requirements. The last component is object-oriented programming which is the process of implementing the software system so it becoming a reality. By the use of these three components, object-oriented development is a beneficial tool in software development. Programmers that use software objects see the benefits that object-oriented analysis, design, and programming have to offer. The software objects can model real-world objects which help programmers to simulate real-world situations. This simulation is accomplished by objects becoming abstractions of their real-world counterparts in which they manage their own state and offer services to other objects (Sommerville 249).

To reach my own conclusions about object-oriented development, I followed through the early processes of analysis and design to create an object-oriented model for an accounting financial statement, the balance sheet (Appendix A, Hermanson 12). First, I had to define the objects and the attributes and operations associated with them that

were relevant in creating an object-oriented model of the balance sheet. Next, by using C++, an object-oriented language, I programmed two models. One of the models was done without inheritance, a characteristic of object-oriented programming, and the other model was with inheritance.

Background on Object-oriented Development

The first question that could be asked in discussing object-oriented development is: "What is an object?" There are two components which define an object which are a state representing a set of attributes associated to an object and a set of operations used to modify the object's state. Objects are developed through the use of object-class definition. An object class definition states all the attributes and services which can be associated with an object of a class (Sommerville 250).

One of the difficulties with object-oriented development is determining which objects together can develop a software system (Sommerville 255). There are many proposals for deciding on the objects for a system. One method is grammatical analysis of the system by describing it in a natural language. In this case, objects and attributes are nouns, whereas operations or services are verbs. Another method would be to have software objects represent tangible entities in the application domain. Examples of tangible entities would be aircraft, roles such as an employee, events such as a request, interactions such as meetings, or organizational units such as departments. Also, a scenario-based method can be helpful in defining objects. In this method, software developers define and analyze various scenarios of a system in order to identify objects, attributes, and operations (Sommerville 256).

There are four characteristics that are needed for a program to be object-oriented. First of all, a program must have a set of objects. Each object is accountable for a certain process which can be an operation or set of operations. Also, objects are considered instances or variables of certain classes (Booch 38). Objects and classes are tightly linked together. A class contains the information needed to build and use the objects. The final characteristic is if a program in execution uses classes then they must be members of a hierarchy of classes that are tied together by inheritance relationships. A program can look as though it is object-oriented but if one of these four characteristics is missing then the program falls short of being truly object-oriented (Booch 38).

Before object-oriented programming, the way of developing software systems was less desirable due to the lack of efficiency. Specifications for a programs were revised many times before they could be translated into commands that were understood by the computer. However, this type of programming caused confusion because by the time the program was completed the original idea had a probable chance of being distorted (Verity 93).

Object-oriented development guides programmers towards a new way of thinking about software problems and possible solutions. Now, programmers can use object abstractions to create programs (Booch 39). Software objects can either copy the characteristics and behave in some abstract sense as something physical such as a person or something virtual such as a checking account (McClure 56). Also, the object abstraction describes the essential characteristics of an object that differentiate it from all other objects which clearly states the conceptual boundaries of them (Booch 41). The use of objects and abstraction in object-oriented programming is how programmers can

simulate real-world situations (Budd 15). The abstraction allows programmers to view objects and describe how they act with one another just like real-world entities. The benefit from object-oriented development is that the objects will correspond more closely to the original design created before the implementation phase (Booch 39).

The design phase of a software system is crucial in maintaining and efficiently developing it. Programmers analyze and design solutions to problems in three stages. First, the programmer studies and tries to understand the problem. It is impossible to effectively design a software system without a clear understanding of the problem. It is important to identify a number of solutions and critique them all. The last stage is to be able to describe each abstraction used in the solutions (Sommerville 210).

There are several characteristics in developing the object-oriented model in object-oriented design. For instance, objects are responsible for maintaining their own states and offering services to other objects. Objects are independent entities in which they can be modified because their state and representation information is stored within the object itself. The functionality of the system is described in terms of the objects' operations and services. Also, shared variables are non-existent. Objects talk to each other by calling on services offered by other objects instead of using shared variables. The last characteristic is that objects can be distributed and executed either sequentially or in parallel (Sommerville 248).

There are several benefits for using object-oriented programming which makes it a popular programming technique. Programmers who use object-oriented programming notice an improved reliability (McClure 55). Objects separate program functions from each other allowing a programmer to modify one without disrupting other areas of the

program (Verity 93). Also, increases in productivity can be attributed to the natural correspondence between the program's objects and the real world objects. Clarity and understanding occur because data and programs are stored together (McClure 55).

There are a few drawbacks in using object-oriented programming. Productivity gains from program modularity and reusability can be difficult to accomplish in large complex systems. Programmers building a complex system find challenges in developing the correct objects and choosing the optimal behaviors for objects. For example, Smalltalk, an object-oriented language has the possibility of five thousand different behaviors in its programming environment. For this large environment, programmers need tools to locate certain behaviors. Traditional programmers do find object-oriented programming to have a steep learning curve. Another problem is an insufficiency of any standard object-oriented development methodology (McClure 57).

Background on Inheritance

An important characteristic of object-oriented design and programming is the use of inheritance. Inheritance is a property associated with the objects of a child class (or subclass). Inheritance provides the capability of a child class to access both the data and behavior (operations) of the parent class (or superclass). A subclass will contain all the properties that are associated with the parent class as well as its own properties (Budd 85). Inheritance provides two mechanisms for software programmers that use object-oriented development. One mechanism is that inheritance provides an abstract mechanism allowing a programmer to classify entities in system models. The other mechanism is a reuse mechanism which allows for quick changes to be made to an object without corrupting other parts of the system (Sommerville 254).

To provide direct support for inheritance, an object-oriented language must support the "is a" relationship. Some examples of "is a" relationships are: a red rose is a flower, a dog is a mammal, or a car is a vehicle. Each of these concepts is an object within different classes. The "is a" relationship shows how the first concept (red rose) is a specialized object of the second concept (flower). Also, the behavior and data belonging to the more specific concept form a subset of the behavior and data of the more abstract concept (Budd 31-32). Classes are linked together by "is a" relationships to form a hierarchy of abstractions. These links allow the programmer not to start classes from scratch but to inherit behavior from other classes (Booch 59).

There are two different ways that inheritance can view class hierarchy. The first way is viewed in the object-oriented languages of Smalltalk and Objective-C where the classes form one large inheritance structure. A portion of the Smalltalk class hierarchy is in Appendix B. An advantage of having one large hierarchy structure is a class at the top of the structure can be inherited by all other classes in the hierarchy. This advantage can be accomplished by polymorphism which is another property of inheritance. An example of polymorphism is when the Object class holds the message print then all other classes could be able to respond to this command. The other way to view the class hierarchy is by other object-oriented languages such as C++ and Object Pascal. These languages do not allow classes that are distinctly different to be represented in the same class hierarchy but instead create several smaller class hierarchies with related classes. These small class hierarchies allow a program not to be constrained by holding large class hierarchies when only a few classes are used(Budd 96).

There are many different types of inheritance. First of all, a programmer can use public inheritance which is sometimes referred to as inheritance of the interface (Levin 575). The interface of a class holds the documentation of how to use the class and the public sections of the class operations. In public inheritance, the public sections of the parent class are considered the public sections of the child class. Public inheritance allows classes to mimic generalization (Papurt 326). The child class uses generalization by developing a more general object from the operations from the parent class (Budd 92). For example, a parent class Window could simply define the background to be black-and-white. A programmer could then create a child class ColoredWindow that would inherit the public operations of the Window class but add additional information to have different colors for the background that would replace the black-and-white background (Budd 93).

Another type of inheritance is private inheritance which is also referred to as inheritance of implementation. Inheritance is used in implementation situations when creating a new child class that needs properties from a parent class. The reason for using private inheritance is usually to possibly save coding efforts, storage space, or execution speed. Some variations of implementation inheritance include cancellation, optimization, and convenience. In cancellation, the parent class limits the inherited behavior by choosing certain inherited properties to be unavailable. Optimization is when the child class takes specific implementation information from the parent class to improve its own code. Convenience means that the child class is made a subclass of the parent class because it provides desired properties for the child class (Taivalsaari 448).

So far the inheritance I have described has been simple inheritance where one child class inherits from one parent class. Another type of inheritance is multiple inheritance where the child class inherits behavior from two parent classes. Multiple inheritance can be used to merge disjoint or independent classes (Hollub 226). It is also known as an effective and flexible way to mix capabilities from other classes into one class (Levin 575).

Programmers face many challenges when using multiple inheritance. One difficulty that is attributed to multiple inheritance is name ambiguity. This occurs when a child class has inherited from two parent classes which happen to have the same name for operations. The child class will not be able to distinguish which operation it is calling from the two parent classes if they have the same name. One possible solution is for the child class to change the names of operations with the same name it is invoking from the parent classes. However, the child class can be more specific with the functionality of the member operations in the parent classes (Holub 224). Another difficulty with multiple inheritance is when a class inherits behavior from two other classes which inherit their own behavior from one base class. This difficulty results in a diamond-shaped class hierarchy where a child class D has two parent classes B and C which both have the same parent class A. The problem for class D is that it might inherit twice (or more) from class A. There are three approaches to handle this repeated inheritance. An easy approach used by Smalltalk and Eiffel is to treat repeated inheritance as an illegal operation. Second, C++ allows parent classes to be duplicated, but only if fully qualified names are used to acknowledge the operations of a specific object. The third approach is to view multiple references to the same class as denoting the same class. C++ allows this last

approach by classifying the repeated parent class A to be a virtual class. A virtual class exists when a child class calls a parent class and refers to it as virtual which indicates a shared class. The use of the virtual class means that class D will only acknowledge a class A object once even though it can inherit it from class B or C (Booch 126).

Inheritance provides benefits for object-oriented programming. One of the most recognized benefits is reusability. When behavior is inherited by a child class from a parent class, the code for this behavior does not have to be rewritten in the child class. This reusability helps a programmer to become more productive by saving time in writing large sections of code. By reusing code, inheritance provides greater reliability by continually checking for errors in the program (Budd 86). Also, child class can take advantage of using new versions of the parent class.

Reusability does not come without some weaknesses. For example, it is not always good to reuse code in order to make changes in existing classes. If several child classes are making changes to one parent class could it increase the difficulty of understanding the parent class (Budd 96). Also, reuse in the software industry seems to be on a more individual scale. Programmers are developing their own libraries containing classes to reuse in their development of applications. There are few development organizations that can successfully have reuse on a broader scale (Maring 35). One reason is reusing classes is not being done by a lot of programmers as expected. An example from Andre Casssulo, principal systems integrator at Florida Power & Light Co. who noted that "after five years with objects, we are now putting together processes so that we are actually reusing components." Cassulo also described that the components being reused are common to every application such as a print management or reporting

system. A reason for the slowness in the construction of challenging object models is the difficulty of defining standard objects that can be used throughout a company. Reusability has been slowly accepted because reuse creates a lot of initial work. In the design phase, programmers have to plan out how to reuse classes because it does not happen just by accident which was believed by programmers in the past (Hayes 62).

Another benefit of inheritance is that it can be used as an organizational tool. Inheritance can be used to organize collections of code and design information. For example, if a parent class, Window, which was inherited by a child class, ScrollableWindow, and TextEditWindow was a child class of ScrollableWindow(Levin 570). These classes create levels in a hierarchical tree whereby the ScrollabeWindow and TextEditWindow will add there own additional information to create a more specific window but also hold the behavior of a normal window from the class Window. By organizing these classes into a hierarchical tree it will be easier to understand their relationships with each other. If there is good organization in a class hierarchy then a programmer should not have to look at all classes but the class hierarchy should tell the programmer where to find a particular one that they are looking for (Levin 578).

There are some disadvantages sometimes in using inheritance. One disadvantage is that inheritance can sometimes violate encapsulation which is a property of object-oriented programming. Encapsulation is a programming feature in languages such as C++ that allows access of data only in a strictly defined interface (Wolfsthal 87). The strict interface for C++ is an object's attributes and operations are only visible to the user if they are located in the public section (Wolfsthal 86). There are three ways that inheritance violates encapsulation. Barbara Liskov describes these three violations by

stating, "the child class might access an object variable of its parent class, call a private operation of its parent class, or refer directly to parent classes of its own parent class (cited, Booch 62)." However, object-oriented languages such as C++ provides ways to get around these violations. In C++, an object's attributes and operations can be put into a protected section allowing a child class to access these attributes and operations (Booch 105).

Background on the Balance Sheet

The financial accounting statement, the balance sheet, was useful to show me the benefits of using object-oriented development in designing a software program. The balance sheet presents the financial position of a company by stating its investing and financial activities for a given period (Stickney 7). There are three major components that make up the balance sheet. One component is the assets which provide a picture of the investments of a business. The other two components are the liabilities and shareholders' equity which show how the assets are financed (Stickney 36).

The categories of assets and liabilities can be broken down further into current and noncurrent ones. Current assets are the assets a business plans to convert into cash, sell or consume within one year from the date of the balance sheet. Examples of current assets are cash, accounts receivable, and inventory. Current liabilities are liabilities that a business will pay within one year. Examples of current liabilities are notes payable to banks, accounts payable, salaries payable, and taxes payable. Noncurrent assets are those which will be held by the business for several years such as land, buildings, and equipment. Noncurrent liabilities and shareholders' equity are a business's sources of long-term funds which (Stickney 8).

In preparing the balance sheet, the effects of each transaction are traced. The different kind of sub-components, the assets, liabilities, and shareholder's equity categories are accounts. An account provides the accumulation of the increases and decreases from the transactions which is the current balance(Stickney 47). Because of the nature of the balance sheet each transaction has a dual effect. This means that every transaction will affect at least two accounts. There are four different dual effects that a transaction can have on balance sheet accounts(Stickney 46):

- It increases both an asset and a liability or shareholders' equity.
- It decreases both an asset and a liability or shareholders' equity.
- It increases one asset and decreases another asset.
- It increases one liability or shareholders' equity and decreases another liability or shareholders' equity.

Debits and credits is the way that accounting handles the increases and decreases to the balance sheet accounts. A debit means an increase to an asset account, on the other hand a debit to the liability or shareholders' equity accounts means a decrease to these accounts. A credit describes a decrease in an asset account, but to liability or shareholders' equity accounts a credit is a decrease to these accounts (Stickney 50).

The Balance Sheet Object-Oriented Models

My goal in designing two balance sheet models, one without inheritance and one that used inheritance, was so I could derive my own conclusions that supported the research on how object-oriented development is useful in building software systems. Also, I wanted to gain the knowledge to be able to assess other structures that could be used to implement a balance sheet. The focus of my programming was not to be able to

have a working implementation of a balance sheet. However, I wanted to reach a syntactical correct version to provide myself with another learning tool for developing my knowledge further on using object-oriented development.

Following the steps of object-oriented development, I began by analyzing the application domain of my project. I tried to figure out what happens through the process of creating the balance sheet. What objects were needed to design the object-model for a balance sheet? The method I used to develop the objects was a grammatical analysis of the balance sheet. From my previous knowledge about a balance sheet, I decided the relevant objects, nouns from the balance sheet, would be an account, transaction, and the debit or credit. For the model without inheritance the debit and credit became one object which I called debitcredit. When I moved onto the model with inheritance, I still had the account, transaction, and debitcredit as objects in parent object classes. However, the inheritance provided me the tool to have more specific child objects which for an account were: asset, liability, and shareholders' equity objects as well as taking the debitcredit object and forming a debit or credit object.

After I knew what objects were going to be used, my next two major decisions were to define the attributes and operations for each object. I did not have to define all of the attributes and operations for the specifc child objects(asset, liability, shareholders' equity, debit, and credit) because the attributes and some of the operations were inherited from the parent objects. The only operations that were not inherited by the child objects were the ones specifically defining its behavior.

The method I used to decide the attributes and operations for an object were by thinking of the role each of them played in the balance sheet as a whole. The attributes for an account object are:

- account identification number
- type (asset, liability, or shareholders' equity)
- account name
- initial balance
- current balance
- a list of debits and credits associated with an account.

For the functionality of the account object, the operations included being able to construct an account, return the attributes, and move through the list of debits and credits. The list of attributes was simpler for a transaction than an account. For example, attributes for a transaction are:

- transaction identification number
- date of the transaction
- a list of debits and credits associated with an account.

I decided that the operations for a transaction would be to construct a transaction, return the attributes, and traverse the list of debits and credits. For a debit and credit, the attributes are:

- account identification number associated with it
- transaction identification number
- amount.

Functionality included with a debit or credit would be to construct one and return the attributes. When I used inheritance, I did have to add a few operations for a debit or credit which were how these two objects affected the different types of accounts.

My next step in following object-oriented development was to figure out possible solutions in creating the balance sheet's object-oriented model. The first solution was a

debit and credit based structure (Appendix C, C3) that would include account, debit, and credit objects. Each account would be associated with their correct debit or credit objects. A debit or credit based approach would make it easy to create accounts, debits, and credits. Also, it would be easy to search for accounts or debits and credits associated with an account. However, it would not be easy to create transactions because no object would be designed specifically for transactions. Searching for transactions or debits and credits associated with a transaction would be quite difficult. One major drawback for the debit and credit approach is that it does not model the real-world situation of a balance sheet which was why I was using object-oriented development.

Another approach to designing the object-oriented model is an account based structure (Appendix C, C1). The only objects that would be defined is an account. Each account object would be able to control a list of debits and credits. In order to create transactions, the programmer would accomplish this task from the account object class. The same benefits and drawbacks for the debit and credit based structure applied to the account based structure.

Instead of looking from the perspective of an account it is also possible to create a structure from the transaction point of view (Appendix C, C2). The transaction structure would include transactions, debits, and credits. Each transaction holds a list of debits and credits that are associated with it. The entries of debits and credits theoretically points to account entries in a list of all the accounts. By using the transaction perspective, it would be easy to create transactions. Programmers using this approach will be able to search for transactions and debits or credits associated with a transaction. However, it would be difficult to search for accounts or specific debits and credits

associated with accounts. Also, it would not be easy to create accounts. The transaction approach does not solve the problem from the debit and credit or account based structure of being able to model the real-world situation of the balance sheet.

The last approach I came up with was combining the account and transaction based structure (Appendix C, C4). For this last structure, the objects are accounts, transactions, debits, and credits. These objects refer to all the real objects that are a part of a balance sheet. The account and transaction objects would have the functionality to modify their own list of debits and credits that are associated with them. This approach becomes quite easy to use in creating and searching for all the objects. Also, a combined account and transaction approach models more closely the real-world situation of the balance sheet.The only drawback is that efficiency is compromised because each account and transaction object will have a debit and credit list. For each object, this list could have a variable length size.

I decided on using the combined account and transaction structure for constructing the two object-oriented model using an object-oriented language, C++. One of the main reasons for picking the model I did was the combined approach created software objects for all of the real-world components of a balance sheet. By including all the real-world components as objects in the model, I was able to simulate closely the real-world situation of the balance sheet than any of the other possible structures. Another reason for picking the combined structure is that it would be easy to create and search for accounts, transactions, debits, and credits.

Once I knew that I was going to use the structure that combined the account and transaction approach, I was able to start contstructing the two models. I began with the

object-oriented model without inheritance and then created the object-oriented model with inheritance. There were two directories containing files that were used in creating the object-oriented models. One directory was for files that used no inheritance (Appendix D) and the other one was for files that used inheritance (Appendix E).

I made several decisions before starting to program the object-oriented model with no inheritance. First of all, what object classes would I use in this implementation? The object classes that I decided on were account, transaction, debitcredit, and balance sheet. The reason for a debitcredit class was because debits and credits have the same attributes and I felt I could work around the differences they had in functionality. The only functionality difference for a debit and credit is how they affect accounts. Another decision was what criteria I would allow a user to search for in the balance sheet. The balance sheet object class contained the implementation for the searching functions. There were five possible search functions:

- searching for an account by the account identification number
- searching for an account by the account name
- searching for a transaction by the transaction identification number
- searching for a debit or credit by the account identification number and the identification number for the debit or credit
- searching for a debit or credit by the transaction identification number and the identification number for the debit or credit.

Also, I assured myself that using lists was the best data structures for me to use in keeping track of accounts, transactions, debits and credits. The reason for using lists is they are easy to implement and understand compared to other more complex data structures.

Creating an "objects" file (Appendix D, D10) became quite beneficial. The enum statement was used for stating the types of accounts (asset, liability, or shareholders' equity) and debitcredts (debit or credit). Also, included were mini object classes that I

felt would be used throughout the other files. The objects included within these classes were minor compared to other objects such as an account in simulating the real-world situation of an object. One of the most important class in this "objects" file was the list class. The functions included in the list class are a list constructors, returning if the list was empty, inserting an element in a list, moving forwards or backwards in the list, and stopping at certain positions in the list. Including these functions was helpful because they were used throughout many other classes such as account or transaction classes which had lists as attributes.

The functionality implemented in each object class was the operations that I had defined for each object in the beginning phase of my object-oriented development. The balance sheet object class had more functionality as a whole because it would be the interface to incorporate the use of all the objects as components of a balance sheet. I did not include the construction of a balance sheet object because it was not necessary due to the balance sheet using other objects. The balance sheet object class worked with and modified three lists of accounts, transactions, and debitcredits. There were functions to check if the lists were empty and move throughout each list provided by the objects file.

Other functions included in the balance sheet class were to insert accounts, transactions, debitcredits into their perspective lists (Appendix D, D4). These three functions helped me to view the process of having a transaction, figuring out the correct debits and credits, and finding out how an account's current balance would be affected by these debits and credits. It was relatively easy to implement the function to insert an account or debitcredit because it was just a call to the insert function from the objects file to insert a new account or debitcredit into their lists.

The function for the insertion of a transaction was more complicated than inserting an account or debitcredit (Appendix D, D4). I first had to decide which parameters I needed the user to pass in. I chose the date, identification number for a transaction, and list of debitcredits associated with the transaction. For each debitcredit their was a list of steps I needed to accomplish. First of all, I needed to construct a debitcredit object which would be constructed from the information from the specific debitcredit I was acquiring from the list. There were two searches I made. One search was to get the account identification number from the debitcredit and use it to search for the account it belonged too. After my searches were completed and I received the correct information, I was able to insert the debitcredit into the list of all debitcredits. Next, I called the function UpdateAcct from the debitcredit class and I passed to it the debitcredit object. This function was used to modify the current balance of the account depending on what type the account and debitcredit were. I originally created a list containing the identification numbers for the debitcredits this was easy to do by using the copy constructor from the list class in the objects file. One other step was to insert the identification number of the debitcredit into this new list. The final step was to create a transaction object and insert it into the list of all transactions. All these steps closely followed the process of how a transaction updates accounts using debits or credits in the real-world situation of the balance sheet.

The next phase of my programming was to implement the object-oriented model with inheritance. It surprised me how easy this was to do because most of the functionality was already in the files that did not use inheritance. I was able to reuse the Balance Sheet class, objects classes, and the transaction class files. The purpose of using

inheritance was to be able to define the objects more specifically to model their real-world counterparts. For the programming done in these files with inheritance, I had to decide which extra object classes I needed to maintain the child objects. I chose for an account object to have child classes for assets, liabilities, and shareholders' equity and the account class would remain the parent class. Also, I defined an object class for a debit and a credit with the debitcredit class being the parent class. The child object classes were quite simple to implement. Their functionality was inherited except for updating accounts which was different for each class. For each of these child object classes, I did need to create the constructor because I wanted to call the parent class constructor with particular parameters. The account, debitcredit, and transaction class mostly remained the same from the files without inheritance with a few modifications for updating accounts. I also continued using the objects file throughout my implementation with inheritance.

There were some specific decisions I made for the object-oriented model with inheritance. I decided to protect some of the attributes such as the current balance and the list of debits and credits identification numbers associated with an account. The reason to protect these attributes was so that the user could not directly affect them.

Another decision I made was how to take advantage of inheritance in the updating of accounts. Updating an account started in the function for inserting a transaction in the balance sheet class (Appendix E, E4). The call in the balance sheet class was to a virtual update account with a debitcredit object as the parameter from the account class (Appendix E, E6). A virtual function in a parent object class does not have a code section because it is within the child object class. Each child classes included inserting

the identification number into a list of debit and credit identification numbers for a particular account (Appendix E, E8-E10). Also, I modified the current balance by calling a virtual function in the debitcredit class (Appendix E, E11). The functionality could be found for this virtual function in the credit or debit class where there was a function definition for each type of account returning the balance after subtracting or adding the amount depending on if it was a debit or credit (Appendix E, E13-E14).

Conclusions about Object-Oriented Development

There are several conclusions to draw from this project. The most important lesson I learned about object-oriented development was that software objects can take on the characteristics and functionality of their real-world counterparts. In accounting, an account will contain: the account name, the type (asset, liability, shareholder's equity), if it is an asset or liability then is it current or non-current, the debit and credit amounts associated to the account, the initial balance, and the current balance. For my software objects, I was able to construct them with most of the real-world account attributes. The only attribute that I omitted was if the account was an asset or liability then was it current or non-current. The reason for this omission is that distinguishing between current or non-current assets and liabilities is not necessary in the design phase because it only groups certain assets or liabilities together which was not an important feature for my object-oriented models. If I had used object-oriented programming to fully implement the balance sheet, I would have wanted to take into consideration whether an asset or liability was current or non-current. I would want to consider the type of asset or liability in the object-oriented programming phase because it would help me to get closer to what a balance sheet actually looks like. I added one extra attribute to the list of attributes for

a software account object which was an account identification number. I used an account identification number as a programming feature to make it easier to look for particular accounts in a list of account. Also, the software object behaved in the same manner as their real-world counterparts. An account in the real-world situation of a balance sheet updates its current balance by either subtracting or adding the amount of the debit or credit to the current balance depending on the type of account (asset, liability, shareholders' equity). I was able to update my software account objects' current balance in the same manner.

Throughout the designing of my object-oriented models it became apparent that inheritance was helpful in creating software objects that more closely modeled their real-world counterparts. In the model without inheritance, I created general objects of the balance sheet such as an account, a transaction, and a debitcredit object. There was a class which contained the functionality for each of these general objects. However, in my second model the use of inheritance helped me to separate the functionality in these classes for the general objects. In the second model I was able to create more specific objects which had classes with them to hold the specific functionality for those new objects. These specific objects which used inheritance were closer to defining the real-world objects of a balance sheet. I separated the debitcredit and the account objects into their child objects which helped me to attribute more directly the functionality for accounts, debits, and credits. For instance, the debitcredit class held the functionality for both a debit and a credit in how they update accounts. In my object-oriented model with inheritance, I was able to have a debit and credit class that separated the functionality for how a debit and credit updates an account. By separating how a debit and credit

modified an account individually, I was able to model the real world objects more closely.

I gained more efficiency in my second model with inheritance by the elimination of the if-statements which had been used in my classes without inheritance. The reason for this elimination was before these if-statements were used to check if an account was an asset, liability, or shareholders' equity or if a debitcredit was either a debit or credit. Now with inheritance, I did not have to use the if-statements but instead I could program in the asset, liability, shareholders' equity, debit, and credit classes the specific functionality for the child objects (asset, liability, shareholders' equity, debit, and credit).

By using object-oriented development to simulate this real-world situation, I saw how it provided so much clarity and understanding. The first phase of object-oriented development, object-oriented analysis, was a worthwhile experience to go through. In object-oriented analysis, I decided on the overall object to be a balance sheet object which referred to three other objects the account, transaction, debit, and credit objects for information. Once I knew what objects to construct I decided on the attributes and operations to associate with the account, transaction, debit, and credit objects. After ending the first phase, I had a clear understanding of the objects. However, I did not have a clear understanding of how the objects fit together but this came later when I was designing the object-oriented models using C++. In implementing my object-oriented design a function such as inserting a transaction clearly showed the accounting process of how a transaction gets inserted into a transaction list and the debits and credits associated with the transaction affect the current balance of the right accounts. Also, using lists was

a good data structure to clearly show a collection of objects such as accounts, transactions, debits, and credits could be grouped together.

The balance sheet did not have a complex problem domain so I saw how beneficial reuse from using inheritance can be to a programmer. A benefit of reusing code was that I was able to program more efficiently. For example, putting common declarations and classes into the "objects" file was an efficient way to program. There were several lists throughout my files and by keeping the functionality of moving through the list in the "objects" file, I was able to save time in having to code those functions every time I used a list in a file. Also, the balance sheet class, transaction class, and the "object" file from the files without inheritance were all reused again in constructing the object-oriented model with inheritance which saved time in programming.

I noticed by using an object-oriented language I was able to set common design structures throughout the files in both object-oriented models. For instance, each class had a similar constructor in which it set the attributes that were specific to the object of that class. In designing the object-oriented models for the balance sheet there was a balance sheet object and conceptually below that was a layer of all the other objects which were the transaction, the account, and the debitcredit objects. By having my software objects in one layer, I was able to have similar constructors for all the objects. It would become more challenging to keep using the same design structure in classes of a complex system. In a complex system there could be more layers of objects below the overall object. Therefore by having multiple layers of objects it sometimes is not be possible to have the same design structures for the objects in each layer.

One of the final lessons I learned was how when I changed sections of a file it caused modification in other files. For instance, when I changed the functionality in the insert transaction function in the balance sheet that affected other files because this function traced the process of how a transaction updates an account through the classes for transaction, accounts, and debitcredits. Having to modify other files after making changes happens only in the object-oriented design phase. However, in object-oriented programming when you implement the design the functionality is constructed for each specific object individually. Therefore, the software objects in the implementation phase become independent entities because there functionality does not overlap so modifications in one section do not cause other sections of a program to be modified.

Overall Conclusion

Object-oriented development will continue to be widely used in creating software programs. The continued support for object-oriented development is based on object-oriented analysis, design, and programming being able to model and simulate real-world situations. My own conclusions came from the valuable experience of designing my own object-oriented models of a balance sheet. I was able to create software objects such as accounts, transactions, debits, and credits that had the same attributes and functionality as their real-world counterparts. Furthermore, industry researchers believe that object-oriented development has real promise for moving businesses in a positive direction.

Appendices

Appendix A: The Coca-Cola Company and Subsidiaries Consolidated Balance Sheet

Appendix B: Portion of Smalltalk Class Hierarchy

Appendix C: Structures for Object-oriented Model

Appendix D: Code for Object-oriented Model Without Inheritance

Appendix E: Code for Object-oriented Model With Inheritance

Appendix A: The Coca-Cola Company and Subsidiaries Consolidated Balance Sheet

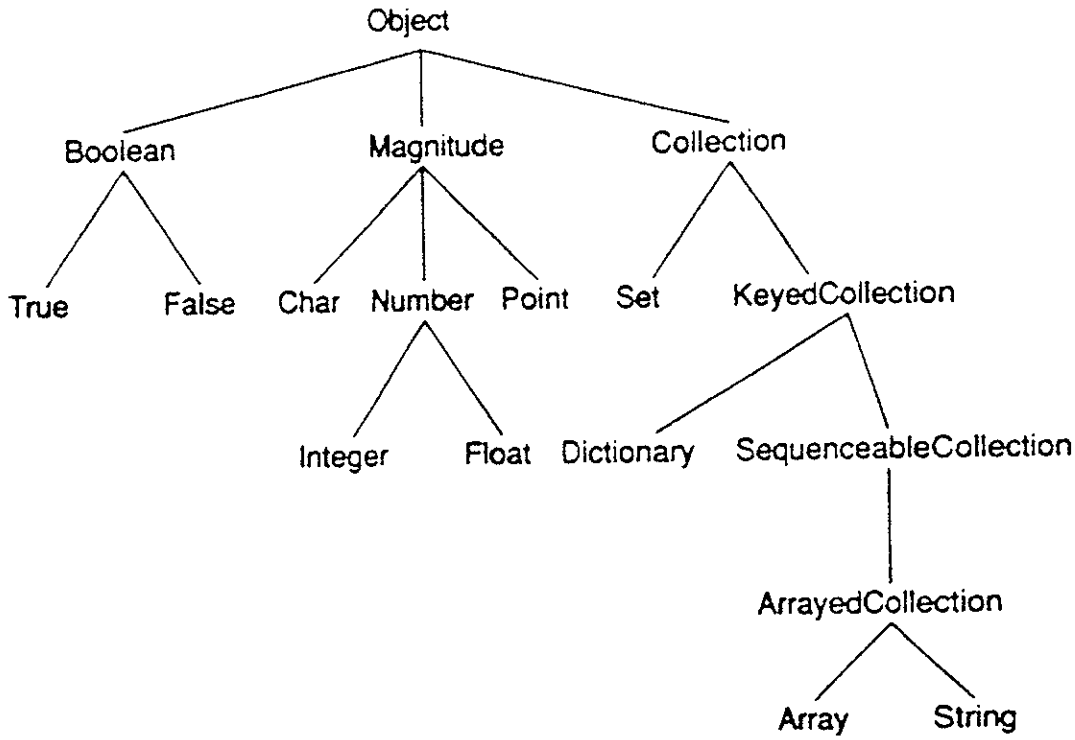
CONSOLIDATED BALANCE SHEETS

December 31, *(In millions except share data)*	1993	1992
Assets		
Current		
Cash and cash equivalents	**$ 998**	$ 956
Marketable securities, at cost	**80**	107
	1,078	1,063
Trade accounts receivable, less allowances of $39 in 1993 and $33 in 1992	**1,210**	1,055
Finance subsidiary receivables	**33**	31
Inventories	**1,049**	1,019
Prepaid expenses and other assets	**1,064**	1,080
Total Current Assets	**4,434**	4,248
Investments and Other Assets		
Investments		
Coca-Cola Enterprises Inc.	**498**	518
Coca-Cola Amatil Limited	**592**	548
Other, principally bottling companies	**1,125**	1,097
Finance subsidiary receivables	**226**	95
Marketable securities and other assets	**868**	637
	3,309	2,895
Property, Plant and Equipment		
Land	**197**	203
Buildings and improvements	**1,616**	1,529
Machinery and equipment	**3,380**	3,137
Containers	**403**	374
	5,596	5,243
Less allowances for depreciation	**1,867**	1,717
	3,729	3,526
Goodwill and Other Intangible Assets	**549**	383
	$ 12,021	$ 11,052

December 31,	1993	1992
Liabilities and Share-Owners' Equity		
Current		
Accounts payable and accrued expenses	**$ 2,217**	$ 2,253
Loans and notes payable	**1,409**	1,967
Finance subsidiary notes payable	**244**	105
Current maturities of long-term debt	**19**	15
Accrued taxes	**1,282**	963
Total Current Liabilities	**5,171**	5,303
Long-Term Debt	**1,428**	1,120
Other Liabilities	**725**	659
Deferred Income Taxes	**113**	82
Share-Owners' Equity		
Common stock, $.25 par value—		
Authorized: 2,800,000,000 shares		
Issued: 1,703,526,299 shares in 1993; 1,696,202,840 shares in 1992	**426**	424
Capital surplus	**1,086**	871
Reinvested earnings	**9,458**	8,165
Unearned compensation related to outstanding restricted stock	**(85)**	(100)
Foreign currency translation adjustment	**(420)**	(271)
	10,465	9,089
Less treasury stock, at cost (406,072,817 common shares in 1993; 389,431,622 common shares in 1992)	**5,881**	5,201
	4,584	3,888
	$ 12,021	$ 11,052

See Notes to Consolidated Financial Statements.

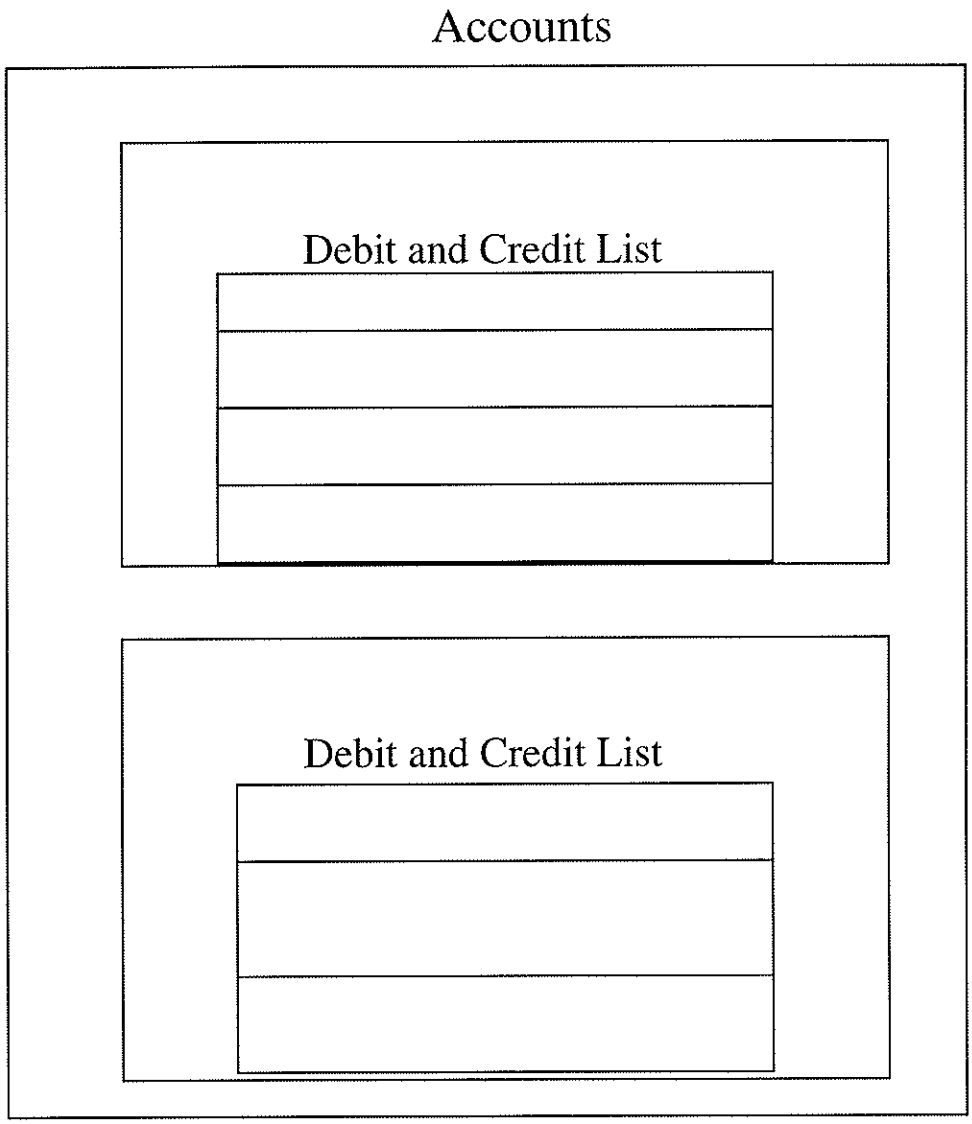
Appendix B: Portion of Smalltalk Hierarchy



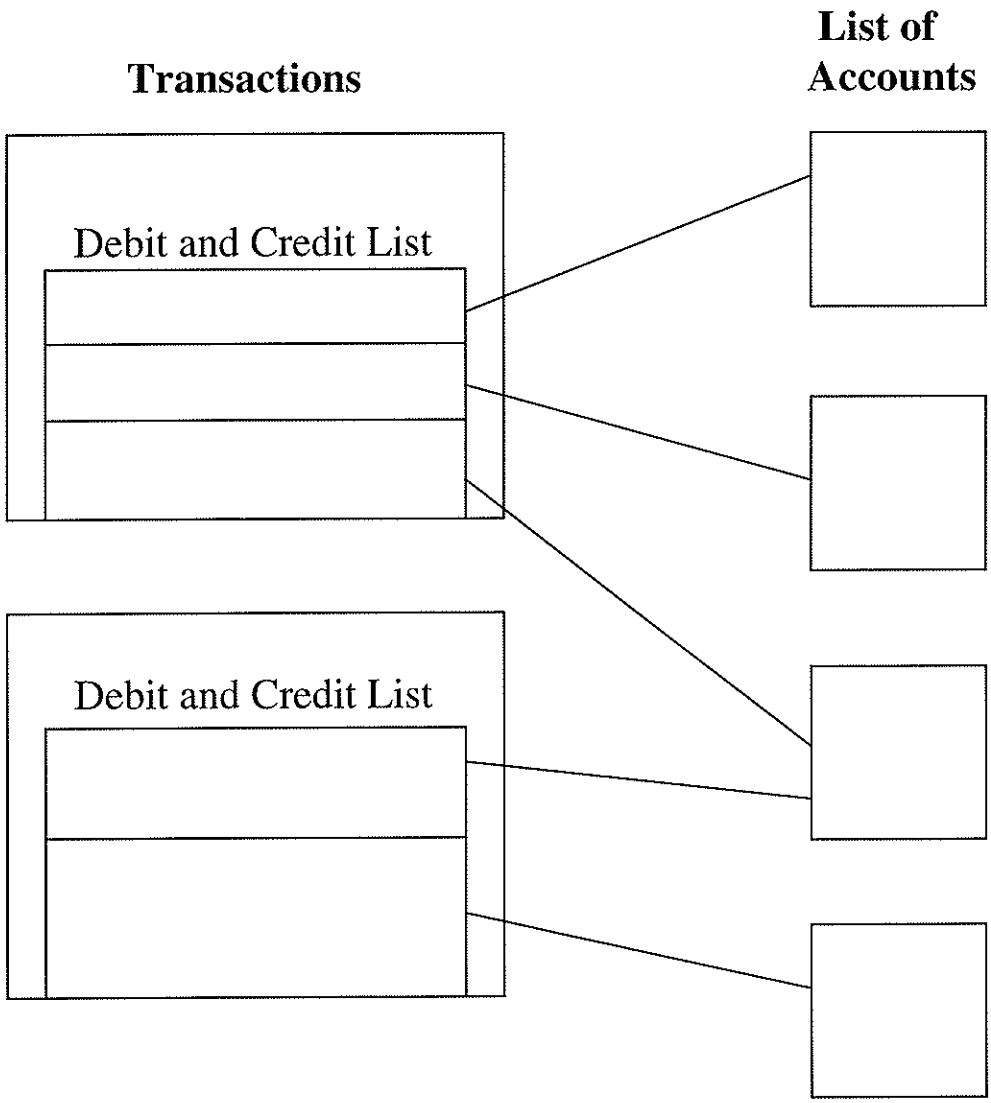
A portion of the Little Smalltalk class hierarchy.

Appendix C: Structures for Object-oriented Models

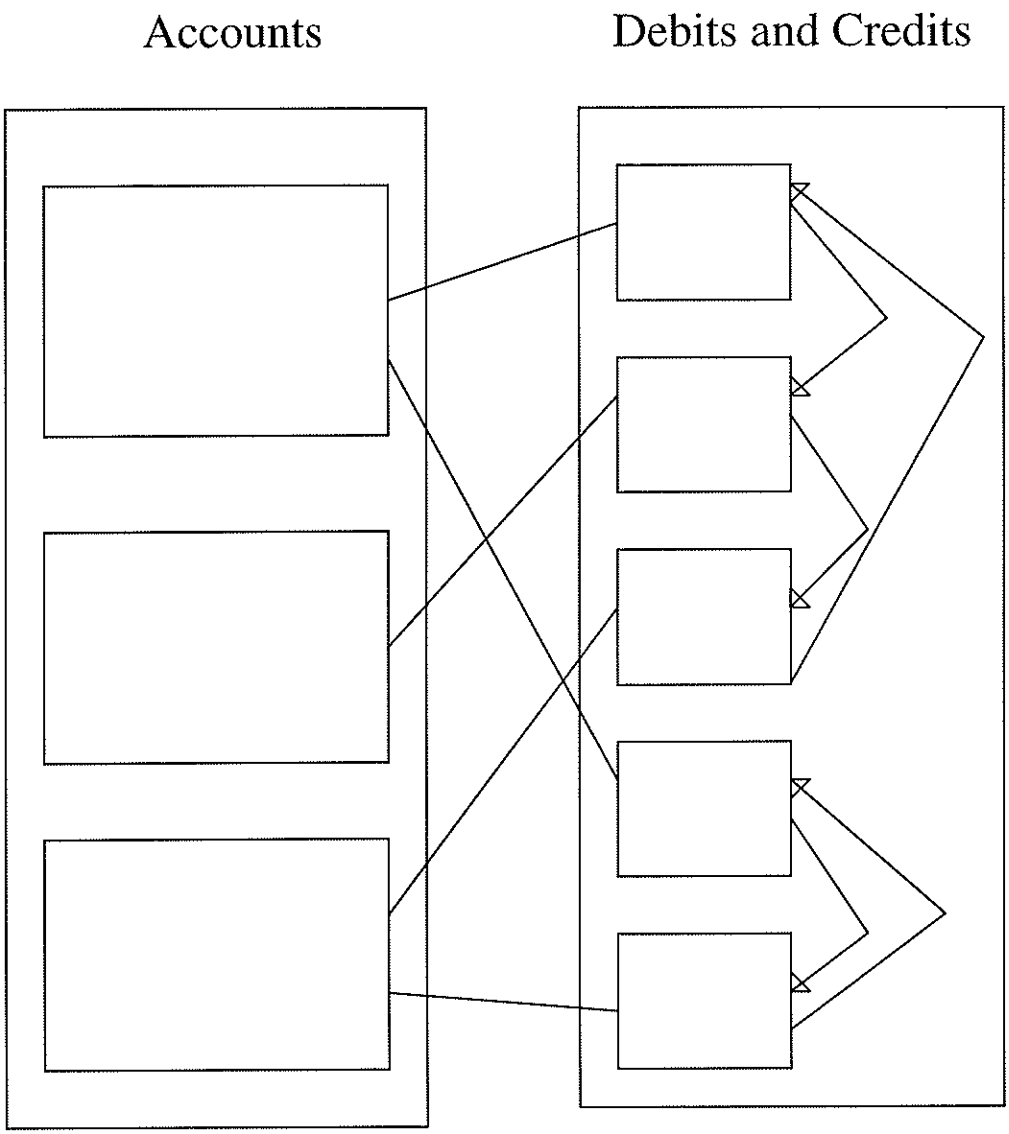
Account-based Implementation



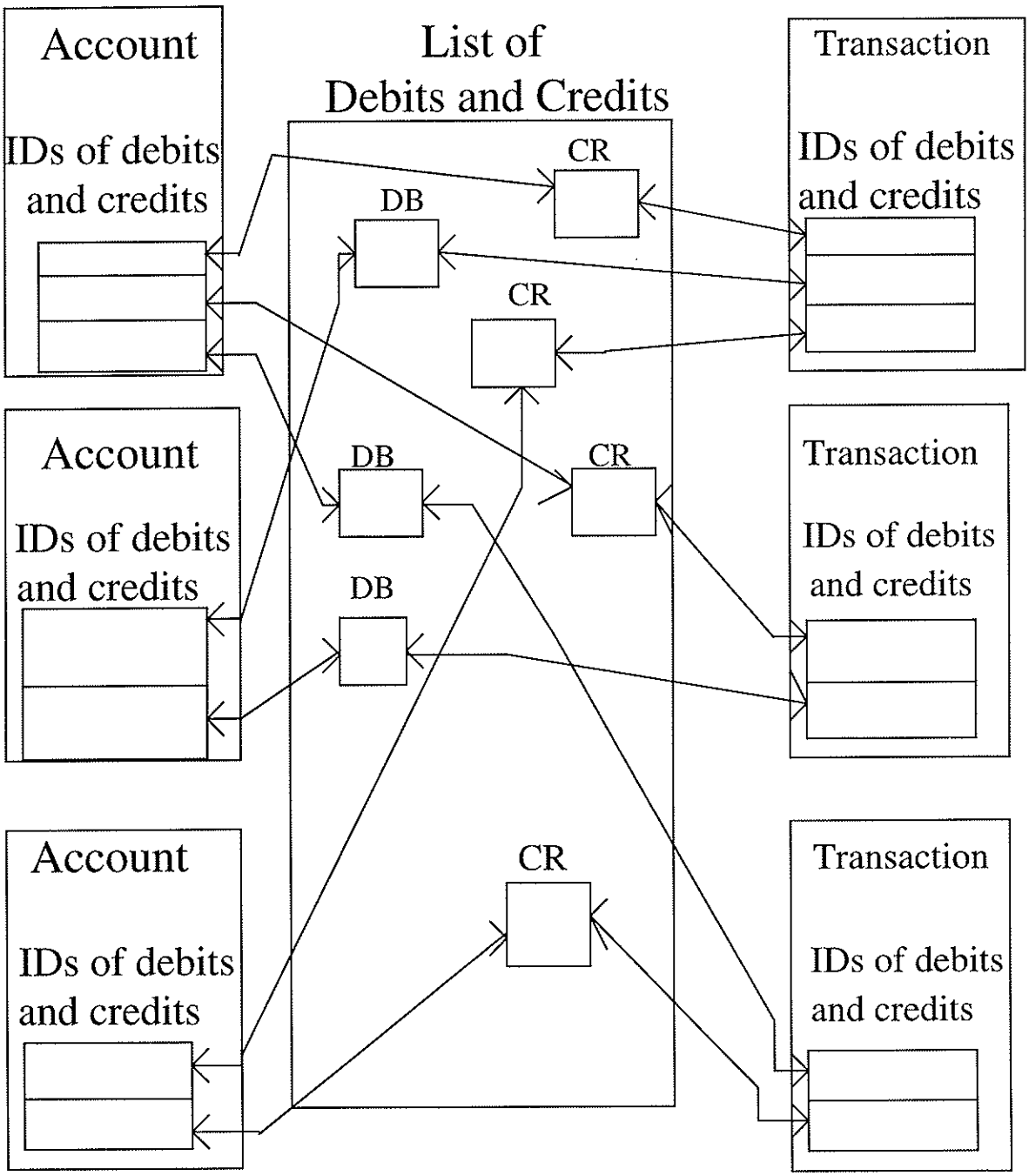
Transaction-based Implementation



Debit and Credit Based Implementation



Combining Account-based and Transaction-based



Appendix D: Object-oriented Model Without Inheritance

//Balance Sheet class declarations for without inheritance and with inheritance

```
#ifndef _BALANCE_SHEET_H_
#define _BALANCE_SHEET_H_

#include <iostream.h>
#include "objects.h"
#include "Account.h"
#include "DebitCredit.h"
#include "Transaction.h"

class BalanceSheet
{
        public:
                // BalanceSheet : default constructor
                //   Pre  : none
                //   Post       : none
                BalanceSheet();

                // ~BalanceSheet : destructor
                //   Pre        : none
                //   Post       : none
                ~BalanceSheet();

                // IsAcctEmpty : observer--is the acctList empty?
                //   Pre     : none
                //   Post    : unchanged
                //   Return value  : true iff the acctList is empty
                int IsAcctEmpty()      { return acctList.IsEmpty(); }

                // IsTransEmpty : observer--is the transList empty?
                //   Pre     : none
                //   Post    : unchanged
                //   Return value  : true iff the transList is empty
                int IsTransEmpty()     { return transList.IsEmpty(); }

                // IsDCEmpty : observer--is the dcList empty?
                //   Pre     : none
                //   Post    : unchanged
                //   Return value  : true iff the dcList is empty
                int IsDCEmpty()     { return dcList.IsEmpty(); }

                // InsertAcct : observer--return a new acctList with the
                //                      added element newAccount
                //   Pre    : none
                //   Post   : unchanged
                 void InsertAcct(const Account & newAccount);
```

```
// InsertTrans : observer--return a new transList with the
//                        added element newTrans
//   Pre    : none
//   Post   : unchanged
 void InsertTrans(date Date, ID Trans,
                        List <DebitCredit> newdclist);

 // SearchAcctID : is acctID in the acctList?
 // Pre:   none
 // Post:  EndofAcctList () || (NextAcct()).GetID == acctID
 void SearchAcctID(ID acctID);

 // SearchAcctName : is the account name in the acctList?
 // Pre:   none
 // Post:  EndofAcctList () ||
 //        (NextAcct()).GetAccountName == acctname
 void SearchAcctName(String acctname);

 //SearchTransID : is transID in the transList?
 // Pre:   none
 // Post:  EndofTransList () || (NexTrans()).GetID == transID
 void SearchTransID(ID transID);

 // Search_DC_Acct : is the debit or credit in the dclist?
 // Pre:   none
 // Post:  EndofAcctList () ||
 void Search_DC_Acct(ID acctID, ID DCID);

 // Search_DC_Trans : is the debit or credit in the dclist?
 // Pre:   none
 // Post:  EndofAcctList () ||
 //
 void Search_DC_Trans(ID transID, ID DCID);

 // MovetoFirstAcct(), NextAcct(), MovetoNextAcct(), and
 // StartofAcctList() are functions to begin moving through the
 // acctList from front to back.
 void MovetoFirstAcct()    { acctList.MovetoStart          (); }
 Account & NextAcct()      { return acctList.NextElement (); }
 void MovetoNextAcct()     { acctList.MovetoNext           (); }
 int StartofAcctList()     { return acctList.StartofList   (); }

 // MovetofLastAcct(), PrevAcct(), MovetoPrevAcct(), and
 // EndofAcctList() are functions to begin moving through the
 // acctList from back to front.
 void MovetoLastAcct()     { acctList.MovetoEnd            (); }
 Account & PrevAcct()      { return acctList.PrevElement (); }
 void MovetoPrevAcct()     { acctList.MovetoPrev           (); }
 int EndofAcctList()       { return acctList.EndofList     (); }
```

```
        // MovetoFirstTrans(), NextTrans(), MovetoNextTrans(), and
        // StartofTransList() are functions to begin moving through the
        // transList from front to back.
        void MovetoFirstTrans()     { transList.MovetoStart          ();}
        Transaction & NextTrans()   { return transList.NextElement ();}
        void MovetoNextTrans()      { transList.MovetoNext           ();}
        int StartofTransList()      { return transList.StartofList   ();}


        // MovetofLastTrans(), PrevTrans(), MovetoPrevTrans(), and
        // EndofTransList() are functions to begin moving through the
        // transList from back to front.
        void MovetoLastTrans()      { transList.MovetoEnd            ();}
        Transaction & PrevTrans()   { return transList.PrevElement ();}
        void MovetoPrevTrans()      { transList.MovetoPrev           ();}
        int EndofTransList()        { return transList.EndofList     ();}


        // MovetoFirstDC(), NextDC(), MovetoNextDC(), and
        // StartofDCList() are functions to begin moving through the
        // crList from front to back.
        void MovetoFirstDC()     { dcList.MovetoStart          (); }
        DebitCredit & NextDC()   { return dcList.NextElement (); }
        void MovetoNextDC()      { dcList.MovetoNext           (); }
        int StartofDCList()      { return dcList.StartofList   (); }

        // MovetofLastDC(), PrevDC(), MovetoPrevDC(), and EndofDCList()
        // are functions to begin moving through the crList from
        // back to front.
        void MovetoLastDC()      { dcList.MovetoEnd            (); }
        DebitCredit & PrevDC()   { return dcList.PrevElement (); }
        void MovetoPrevDC()      { dcList.MovetoPrev           (); }
        int EndofDCList()        { return dcList.EndofList     (); }


    private:
        List <Account>       acctList;
        List <DebitCredit>       dcList;
        List <Transaction>       transList;


};

#endif
```

//BalanceSheet class definitions for without inheritance and with inheritance

```
#include "newBalanceSheet.h"
#include <iostream.h>

BalanceSheet :: BalanceSheet(){
        cout<< "Welcome to the Balance Sheet for Company A" <<endl;
        cout<< "              December 31, 1997            " <<endl;
}

BalanceSheet :: ~BalanceSheet(){ }

void BalanceSheet :: InsertAcct(const Account & newAccount){
        acctList.Insert(newAccount);
}

void BalanceSheet :: InsertTrans(date Date, ID Trans,
                                    List <DebitCredit> newdclist){

        List <ID> newlist;
        for(newdclist.StartofList(); !newdclist.EndofList();
            newdclist.MovetoNext()){
                DebitCredit & dc = newdclist.NextElement();
                ID Acctid = dc.GetAccountID();
                SearchAcctID(Acctid);
                Account & A = NextAcct();
                dcList.Insert(dc);
                A.UpdateAcct(dc);
                newlist.Insert(dc.GetDCID());
        }

        transList.Insert(Transaction(Date, Trans, newlist));
}

void BalanceSheet :: SearchAcctID(ID acctID){

        acctList.MovetoStart();
        while(!acctList.EndofList() &&
            ((acctList.NextElement()).AccountID () != acctID)){
                acctList.MovetoNext();
        }
}


void BalanceSheet :: SearchAcctName(String acctname){
        acctList.MovetoStart();
        while(!acctList.EndofList() &&
            ((acctList.NextElement()).AcctName() != acctname)){
                acctList.MovetoNext();
        }
}
```

```
void BalanceSheet :: SearchTransID(ID transID){
        transList.MovetoStart();

        while(!transList.EndofList() &&
            ((transList.NextElement ()).TransID () != transID))
        {
            transList.MovetoNext();
        }
}

void BalanceSheet :: Search_DC_Acct(ID acctID, ID DCID){
        dcList.MovetoStart();
        while(!dcList.EndofList() &&
            ((dcList.NextElement ()).GetAccountID() != acctID) &&
            ((dcList.NextElement ()).GetDCID() != DCID))
        {
                dcList.MovetoNext();
        }
}
void BalanceSheet :: Search_DC_Trans(ID transID, ID DCID){
        dcList.MovetoStart();

        while(!dcList.EndofList() &&
            ((dcList.NextElement ()).GetTransID() != transID) &&
            ((dcList.NextElement ()).GetDCID() != DCID))
        {
                dcList.MovetoNext();
        }
}
```

```
// Account class declarations

#ifndef _ACCOUNT_H_
#define _ACCOUNT_H_

#include "objects.h"
#include "Transaction.h"

class Account{
        public:
                Account(AcctTypes Type, AccountName Name, ID Acct):
                        type(Type), name(Name), acctID(Acct) {initbal = 0.00;}

                ~Account(){}

                ID AccountID(){
                        return acctID;
                }

                String AcctName(){
                        return (name.GetAccountName());
                }

                String GetType(){
                        return type;
                }

                money CurrentBal(){
                        return currbal;
                }


                // MovetoFirstDC(), NextDC(), MovetoNextDC(), and
                // StartofDCList() are functions to begin moving through the
                // crList from front to back.
                void MovetoFirstDC()   { dcList.MovetoStart          (); }
                ID & NextDC()          { return dcList.NextElement (); }
                void MovetoNextDC()    { dcList.MovetoNext          (); }
                int StartofDCList()    { return dcList.StartofList  (); }


                // MovetofLastDC(), PrevDC(), MovetoPrevDC(), and EndofDCList()
                // are functions to begin moving through the crList from
                // back to front.
                void MovetoLastDC()    { dcList.MovetoEnd           (); }
                ID & PrevDC()          { return dcList.PrevElement (); }
                void MovetoPrevDC()    { dcList.MovetoPrev          (); }
                int EndofDCList()      { return dcList.EndofList    (); }


        private:
                friend class BalanceSheet;
```

```
void UpdateAcct(DebitCredit DC){
        if(type == Asset){
                if(DC.GetState() == Debit){
                        currbal = currbal + DC.GetAmount();
                }
                else//DC.GetState() == Credit
                {
                        currbal = currbal - DC.GetAmount();

                }
        }
        else //((type == Liability) ||
                //(type == ShareholdersEquity))
        {
                        if(DC.GetState() == Debit){
                                currbal = currbal - DC.GetAmount();
                        }
                        else//DC.GetState() == Credit
                        {
                                currbal = currbal + DC.GetAmount();

                        }
        }

}

//data members
ID acctID;
AccountName name;
AcctTypes type;
money initbal;
money currbal;
List <ID> dcList;

};
#endif
```

//DebitCredit class declarations

```
#ifndef _DEBIT_CREDIT_H_
#define _DEBIT_CREDIT_H_

#include "objects.h"

class DebitCredit
{

  public:
        DebitCredit(ID dcid, date Date, DCStates State, ID Acct, ID Trans,
                    money Amount) :
              DCID(dcid), Dt(Date), state(State), acctID(Acct),
                transID(Trans), amount(Amount){}

        ~DebitCredit();

        ID GetDCID(){
                return DCID;
        }

        date Date(){
                return Dt;
        }

        money GetAmount(){
                return amount;
        }

        String GetState(){
                return state;
        }

        ID GetAccountID(){
                return acctID;
        }

        ID GetTransID(){
                return transID;
        }

  private:

        ID DCID;
        ID acctID;
        ID transID;
        date Dt;
        String state;
        money amount;
};
#endif
```

//transaction class declarations for without inheritance and with inheritance

```
#ifndef _TRANSACTION_H_
#define _TRANSACTION_H_

#include <iostream.h>
#include "objects.h"
#include "DebitCredit.h"

class Transaction {
        public:
                Transaction (date Date, ID Trans, List <ID> DCList):
                        Dt(Date), transID(Trans), dcList(DCList){}

                ~Transaction ();

                date Date(){
                        return Dt;
                }

                ID TransID(){
                        return transID;
                }

                // IsDCEmpty : observer--is the DClist empty?
                //   Pre     : none
                //   Post    : unchanged
                //   Return value  : true iff the DClist is empty
                int IsDCEmpty()         { return dcList.IsEmpty(); }

                // MovetoFirstDC(), NextDC(), MovetoNextDC(), and
                // StartofDCList() are functions to begin moving through the
                // crList from front to back.
                void MovetoFirstDC()    { dcList.MovetoStart          (); }
                ID & NextDC()           { return dcList.NextElement (); }
                void MovetoNextDC()     { dcList.MovetoNext           (); }
                int StartofDCList()     { return dcList.StartofList   (); }

                // MovetofLastDC(), PrevDC(), MovetoPrevDC(), and EndofDCList()
                // are functions to begin moving through the crList from
                // back to front.
                void MovetoLastDC()     { dcList.MovetoEnd            (); }
                ID & PrevDC()           { return dcList.PrevElement (); }
                void MovetoPrevDC()     { dcList.MovetoPrev           (); }
                int EndofDCList()       { return dcList.EndofList     (); }

        private:
                date Dt;
                ID transID;
                List <ID> dcList;
};
#endif
```

//object class declarations for without inheritance and with inheritance

```
#ifndef _OBJECTS_H_
#define _OBJECTS_H_

#include "CPstring.h"

enum AcctTypes{Asset, Liability, ShareholdersEquity};
enum DCStates{Debit, Credit};

class money{
        public:
                money(){
                        dollars = 0.0;
                }
                money(float Dollars){
                        dollars = Dollars;
                }
                float getDollars(){
                        return dollars;
                }
                void setDollars(float num){
                        dollars = num;
                }

                friend money operator + (const money & Left,
                                         const money & Right){
                        return(Left.dollars + Right.dollars);
                }

                friend money operator - (const money & Left,
                                         const money & Right){
                        return(Left.dollars - Right.dollars);
                }
        private:
                float dollars;
};

class date{
        public:
                date(){
                        month = 1;
                        day = 1;
                        year = 1997;
                }
                int getMonth(){
                        return month;
                }
                int getDay(){
                        return day;
                }
```

```
                int getYear(){
                        return year;
                }
                void setDate(int Mnth, int Day, int Yr){
                        month = Mnth;
                        day = Day;
                        year = Yr;
                }
        private:
                int month;
                int day;
                int year;
};

class ID{
        public:
                ID(){
                        Ident = 1;
                }
                int GetID(){
                        return Ident;
                }
                void SetID(int ID){
                        Ident = ID;
                }

                friend int operator != (const ID & Left, const ID & Right)
                {
                        return (Left.Ident != Right.Ident);
                }
        private:
                int Ident;
};



class AccountName{
        public:
                AccountName(String AcctName){
                        Name = AcctName;
                }
                String GetAccountName(){
                        return Name;
                }
        private:
                String Name;
};
```

```
template <class T>
class List {
        public:
                List();
                // Post: IsEmpty()

                List(const List <T> & X);

                ~List();

                // IsEmpty : observer--is the List empty?
                // Pre: none
                // Post: unchanged
                // Return value : true iff the list is empty
                int IsEmpty(){return(EndofList() == StartofList());}

                // Insert : observer--return a new List with an added element
                //   Precondition: none
                //   Postcondition: unchanged
                void Insert (const T & newElement);

                // MovetoStart : moves to the first element of the List
                // Pre:  !IsEmpty()
                // Post: StartofList ()
                void MovetoStart();

                // NextElement : observer--returns the next element of the List
                // Pre:  !EndofList()
                // Post: list unchanged
                T & NextElement();

                // MovetoNext : moves to the next element of the List
                // Pre:  !EndofList()
                // Post: list unchanged
                void MovetoNext ();

                // MovetoEnd : moves to the last element of the List
                // Pre:  !IsEmpty()
                // Post: EndofList()
                void MovetoEnd();

                // PrevElement : observer--returns the previous element of the
                //                  List
                // Pre:  !StartofList()
                // Post: list unchanged
                T & PrevElement();

                // MovetoPrev : moves to the previous element of the List
                // Pre:  !StartofList()
                // Post: list unchanged
                void MovetoPrev ();
```

```
        // EndofList : observer--is the List at the last element
        //                          of the list
        // Pre:  !IsEmpty()
        // Post: unchanged
        // Return value: true iff the List is at the end of the list
        int EndofList();

        // StartofList : observer--is the List at the first element
        //                          of the list
        // Pre:  none
        // Post: unchanged
        // Return value : true iff the List is at the start of the list
        int StartofList();

    private:

};


#endif
```

Appendix E: Object-oriented Model With Inheritance

```
// Account class declarations

      #ifndef _ACCOUNT_H_
      #define _ACCOUNT_H_

      #include "CPstring.h"
      #include "objects.h"
      #include "DebitCredit.h"


      class Account{
            public:
                  Account(AcctTypes Type, AccountName Name, ID Acct):
                        type(Type), name(Name), acctID(Acct)
                        {initbal = 0.00;}
                  ~Account(){}

                  ID AccountID(){
                        return acctID;
                  }

                  String AcctName(){
                        return (name.GetAccountName());
                  }

                  String GetType(){
                        return type;
                  }

                  money CurrentBal(){
                        return currbal;
                  }

                  // MovetoFirstDC(), NextDC(), MovetoNextDC(), and
                  // StartofDCList() are functions to begin moving through the
                  // crList from front to back.
                  void MovetoFirstDC()    { dcList.MovetoStart          (); }
                  ID & NextDC()           { return dcList.NextElement   (); }
                  void MovetoNextDC()     { dcList.MovetoNext           (); }
                  int StartofDCList()     { return dcList.StartofList   (); }

                  // MovetofLastDC(), PrevDC(), MovetoPrevDC(), and EndofDCList()
                  // are functions to begin moving through the crList from
                  // back to front.
                  void MovetoLastDC()     { dcList.MovetoEnd            (); }
                  ID & PrevDC()           { return dcList.PrevElement   (); }
                  void MovetoPrevDC()     { dcList.MovetoPrev           (); }
                  int EndofDCList()       { return dcList.EndofList     (); }

            private:
                  friend class BalanceSheet;
                  virtual void UpdateAcct(DebitCredit & DC) = 0;

                  //data members
                  ID acctID;
                  AccountName name;
                  AcctTypes type;
                  money initbal;
```

```
protected:
            List <ID> dcList;
            money currbal;
};

#endif
```

```
//Asset class declarations

#ifndef _ASSET_H_
#define _ASSET_H_

#include <iostream.h>
#include "objects.h"

class Asset : public Account
{

      public:
            Asset(AcctTypes Type, AccountName Name, ID AcctID):
                  Account(Type, Name, AcctID);

            ~Asset(){
                  dbList.~List();
                  crList.~List();
            }

      private:
            UpdateAcct(DebitCredit DC){
                  dcList.insert(DC.GetDCID());
                  currbal = DC.NewAssetBalance(currbal);
            }
};

#endif
```

```
//Liability class declarations

#ifndef _LIABILITY_H_
#define _LIABILITY_H_

#include <iostream.h>
#include "objects.h"

class Liability : public Account
{

      public:
            Liability(AcctTypes Type, AccountName Name, ID AcctID):
                  Account(Type, Name, AcctID);

            ~Liability(){
                  dbList.~List();
                  crList.~List();
            }

      private:

            UpdateAcct(DebitCredit DC){
                  dcList.insert(DC.GetDCID());
                  currbal = DC.NewLiabBalance(currbal);
            }
};

#endif
```

```
//ShareholdersEquity class declarations

#ifndef _SHAREHOLDERS_EQUITY_H_
#define _SHAREHOLDERS_EQUITY_H_

#include <iostream.h>
#include "objects.h"

class ShareholdersEquity : public Account
{

      public:
            ShareholdersEquity(AcctTypes Type, AccountName Name, ID
AcctID):
                  Account(Type, Name, AcctID);

            ~ShareholdersEquity(){
                  dbList.~List();
                  crList.~List();
            }


      private:

            UpdateAcct(DebitCredit DC){
                  dcList.insert(DC.GetDCID());
                  currbal = DC.NewSEBalance(currbal);
            }
};
#endif
```

```
//DebitCredit class declarations

#ifndef _DEBIT_CREDIT_H_
#define _DEBIT_CREDIT_H_

#include "objects.h"

class DebitCredit
{

   public:
            DebitCredit(ID dcid, date Date, DCStates State, ID Acct, ID
                        Trans, money Amount) :
                     DCID(dcid), acctID(Acct), transID(Trans), Dt(Date),
                     state(State), amount(Amount){}

      ~DebitCredit();

      ID GetDCID(){
            return DCID;
      }

      date Date(){
            return Dt;
      }

      money GetAmount(){
            return amount;
      }

      String GetState(){
            return state;
      }

      ID GetAccountID(){
            return acctID;
      }

      ID GetTransID(){
            return transID;
      }

   private:
      virtual money NewAssetBalance(money old_balance) = 0;
      virtual money NewLiabBalance(money old_balance) = 0;
      virtual money NewSEBalance(money old_balance) = 0;


protected:

      ID DCID;
      ID acctID;
      ID transID;
      date Dt;
      String state;
      money amount;

};

#endif
```

```
//Debit class declarations

#ifndef _DEBIT_H_
#define _DEBIT_H_

#include <iostream.h>
#include "objects.h"

class Debit : public DebitCredit
{

      public:

            Debit(ID Dbid, date Date, ID Acctid, ID Transid, money Amount):
                        DebitCredit(Dbid, Date, Acctid, Transid, Amount){}


            ~Debit();

      private:

            money NewAssetBalance(money old_balance){
                  money new_balance = old_balance + Debit.GetAmount();
                  return new_balance;
            }
            money NewLiabBalance(money old_balance){
                  money new_balance = old_balance - Debit.GetAmount();
                  return new_balance;
            }
            money NewSEBalance(money old_balance){
                  money new_balance = old_balance - Debit.GetAmount();
                  return new_balance;
            }
};

#endif
```

```
//Credit class declarations

#ifndef _CREDIT_H_
#define _CREDIT_H_

#include <iostream.h>
#include "objects.h"

class Credit : public DebitCredit
{

      public:
            Credit(ID Crid, date Date,ID Acctid,ID Transid, money Amount):
                        DebitCredit(Crid, Date, Acctid, Transid, Amount){}

            ~Credit();

      private:

            money NewAssetBalance(money old_balance){
                  money new_balance = old_balance - GetAmount();
                  return new_balance;
            }
            money NewLiabBalance(money old_balance){
                  money new_balance = old_balance + GetAmount();
                  return new_balance;
            }
            money NewSEBalance(money old_balance){
                  money new_balance = old_balance + GetAmount();
                  return new_balance;
            }

};

#endif
```

Work Cited

Booch, Grady. Object-oriented Analysis and Design With Applications. The Benjamin/ Cummings Publishing Company Inc. New York, 1994.

Budd, Timothy. An Introduction to Object-Oriented Programming. New York: Addison-Wesley Publishing Company Inc, 1991.

Hayes, Frank. "*The Reality of Object Reuse.*" Computerworld. 6 May 1996: 62-63.

Hermanson, Roger H., James Don Edwards, and Michael W. Maher. Annual Report Booklet to Accompany Accounting a Business Perspective. Chicago: Irwin, 1995.

Hollub, Allen. "*The Power of Inheritance.*" Byte Magazine. May 1993: 221-226.

Levin, Harold D. And Jo Ellen Perry. An Introduction to Object-Oriented Design in C++. New York: Addison Wesley Publishing Company Inc, 1996.

Maring, Blayne. "*Object-Oriented Development of Large Applications.*" IEEE Software. May 1996: 33-40.

McClure, Steve. "*Object Technology: A Key Software Technology for the 90's.*" Computerworld.18 May 1992: 53-61.

Papurt, David M. Inside the Object Model: The Sensible Use of C++. New York: Sigs Book, 1995.

Stickney, Clyde P., Roman L. Weil, and Sidney Davidson. Financial Accounting: An Introduction to Concepts, Methods, and Uses. New York: Harcourt Brace Jovanich Inc, 1991.

Sommerville, Ian. Software Engineering. New York: Addison-Wesley Publishing Company Inc, 1996.

Taivalsarri. Antero. "*On the Notion of Inheritance.*" ACM Computing Surveys. 3 September 1996: 438-479

Verity, John W. And Evan I. Schwartz. "*Software Made Simple.*" Business Week. 30 September 1991: 92-100.

Wolfsthal, Yaron. "*The Road to Effective Software Development.*" IEEE Communications Magazine. April 1994: 84-87.