

State of South Dakota  
Candidate's or Committee's Report of Receipts and Expenditures

179 RECEIVED  
FEB - 1 2002  
S.D. SEC. OF STATE

Candidates and candidate committees: File in the office where you filed your nominating petition.  
PACs, political party, ballot question and other committees: File with Elections Department, Secretary of State's Office, 500 E Capitol Ave, Pierre, SD 57501-5070

See "South Dakota Campaign Finance Reporting Guidelines" for specific instructions on completing this report.

Name of Candidate or Committee: Lake County Republican Central Committee

Complete Mailing Address: 23853 461st Ave, Wentworth, SD 57075

Name of Person Making Report: Dave Daniel

Daytime Phone Number: (605) 489-2262

If you are a candidate, what office are you seeking: \_\_\_\_\_

If you are a ballot question committee, indicate which measure(s) the committee was involved with during the reporting period and whether the measure was supported or opposed.  
\_\_\_\_\_

Type of Report: End of Year

For Reporting Period Ending: December 31, 2001

The following verification must be completed before submitting report.

VERIFICATION OF PERSON MAKING REPORT

I, Dave Daniel (type name), certify that I have examined this report and to the best of my knowledge and belief it is true, correct and complete.

Date: 1/29/02

Dave Daniel  
Candidate Signature or Signature of Committee Treasurer or Chairperson

Revised March 1999

Ver 1.01

Filed this 1st day of February 2002

Joyce Hazelton

SECRETARY OF STATE

## CHAPTER 2:

# Object-Oriented Concepts

The purpose of this chapter is to provide you with an introduction to basic Object-Oriented concepts. This is not, however, a vocabulary-intensive discussion. In fact, we will introduce you to only eight new terms:

<i>Attribute</i>	<i>Class</i>
<i>Encapsulate</i>	<i>Inheritance</i>
<i>Instance</i>	<i>Method</i>
<i>Object</i>	<i>Polymorphism</i>

These terms, plus the concepts they represent, are very adequate to provide the OO background you need to learn Object COBOL.

This chapter introduces a simple credit union application which is the basis for examples throughout this and subsequent chapters. Then the systems development process is reviewed and the motivation for using OO is discussed. Next, a brief history of OO is presented, then OO concepts are introduced.

After studying this chapter you will understand the primary differences between traditional systems and OO systems. In addition, you will be familiar with key OO terms and concepts.

### The Credit Union System

---

The Credit Union operates in a small city and serves approximately 3,000 members. The members are primarily employees of a local manufacturing company. Employee dependents can also be members of the credit union.

Each member is a customer and has at least one account. There are three types of accounts available: checking, savings, and loan accounts. Further, there are two types of loan accounts: automobile loans and home loans. A member can have several accounts. For example a member could have two checking accounts, a savings account, plus an automobile loan and a home loan. Each account will have a separate account number. Members are identified by their social security number.

Customer statements are produced each month that summarize all transactions for each account. The checking account does not pay interest. However, the savings accounts do pay interest which is the same rate for all savings accounts. Interest is computed each month by multiplying the savings account interest rate by the average balance for the month. Each loan account is charged an interest rate which was determined at the time the loan was established. The loan interest is computed by



multiplying the loan's interest rate by the current loan balance. These computations are developed in more detail later.

## **OO is The Same and OO is Different**

---

Some argue that OO is not all that different from traditional development methods, while others believe that OO is totally different. In reality, both positions have merit.

OO appears similar to traditional development methods because it employs some fundamental principles of good software engineering such as decomposing a problem into smaller, manageable modules and restricting data access. OO encourages modularization and *requires* restricted data access. However, we still must write code to define data and we must write code to process that data. In OO terms, the data is called **attributes** and the code is called **methods**.

OO is, however quite different because an *object* becomes a system building block containing both data and code (**attributes** and **methods**)! In contrast, in a traditional system the data is contained in files and we develop programs to access these files. However, an object owns and controls its data. The only way one object can access another object's data is to send that object a **message** requesting the data. The data is effectively hidden or **encapsulated**.

The only way then to access the data in an object is to send a **message** to that object. The message will invoke a **method** to carry out the desired process. An information system then becomes a set of *objects* that interact and collaborate by sending and receiving messages. Traditional systems, in contrast, consist of files and programs that access those files. This is shown graphically in Figure 2.1

Also, Object-oriented Programming Languages (OOPLs) are different. They use different syntax and terminology. Even Object COBOL, while retaining the familiar COBOL vocabulary, contains new syntax to accommodate the object extensions. OO analysis and design also use some unique terminology and notation. Significantly, the distinction between analysis and design becomes somewhat fuzzy in OO.

In addition to OO differences, there are significant ongoing changes in the development environment, such as GUIs, client-server architecture, and the Internet. Although not directly a part of OO, these create new challenges for developers, which further complicates systems development.

Finally, although OO has had some dramatic successes, it is not a panacea. Many of the design and development issues that we encounter in traditional development also exist in the world of OO. We can develop a *bad* system using the OO paradigm just as we can when using traditional structured techniques. OO only *enables* us to build better systems faster; it does not *assure* us that we will.



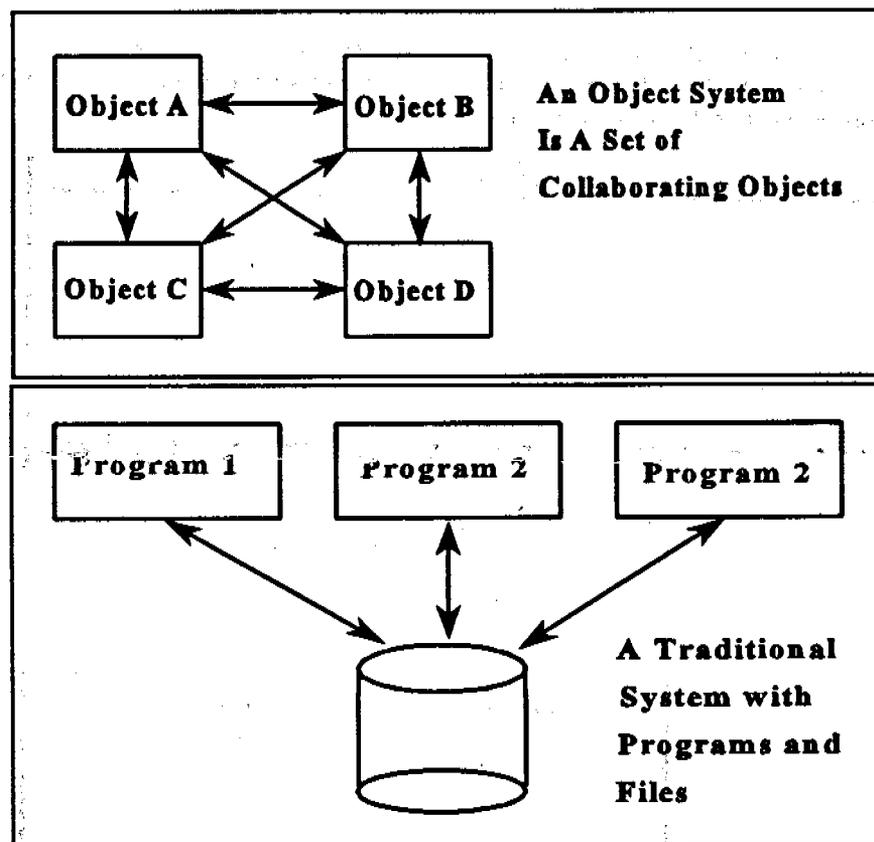


Figure 2.1: Contrasting Object and Traditional Systems

## Software Objects Model Real-World Objects

When designing a software system using the OO methodology, we model *real-world objects* with *software objects*. In Chapter 1 we said the first OOP language was a simulation language (SIMULA). When we build software objects that represent real-world objects, we are actually *simulating* a part of the real world. One can argue that OO development is simply building a simulation model. Real-world objects are all around us. An employee, for example, is a real-world object. A student, a professor, and a customer are all examples of real-world objects. However, real-world objects are not limited to people.

Early in the study of our language, we learned that a noun was a person, place, or thing. A real-world object can be similarly specified. From the problem domain, we identify relevant people, places, and things. *People* can be customers, employees, members, students, and so forth. Examples of *places* are department, region, building, office and room. *Things* can be tangible such as airplane, computer, statement, invoice, and transcript, or less tangible such as account, transaction, and reservation. Incidentally, one reason that OO works well with GUI applications is because the GUI windows and their components are *things* that can be modeled as objects. These GUI objects then interact with other system components.



Referring to our earlier credit union example, there are several real-world objects we can identify: member, customer, teller, account, transaction, checking account, savings account and loan account. Let's consider one real-world object from the example: a credit union customer. There are two characteristics of a customer we want to model: the things a customer *knows* and the things a customer *can do*. For example, a customer *knows* their name, address, phone number, and social security number. Things a customer *can do* include, move (change addresses), and change phone number.

We can model this real-world customer object as a software object named **Customer** as shown in Figure 2.2. The **Customer** object will *know* its Name, Address, Telephone number, and Social Security Number. In addition, **Customer** will be able to *do things*: Change-Address and Change-Telephone-Number.

The things **Customer** *knows* are called **Attributes** and the things it *can do* are called **Methods**. Our software object then has the following Attributes and Methods:

In reality, our system will have many customers, and therefore, many objects. In fact, we will have one software object for each customer. If we have 3,000 real-world customers, then our system will have 3,000 customer software objects.

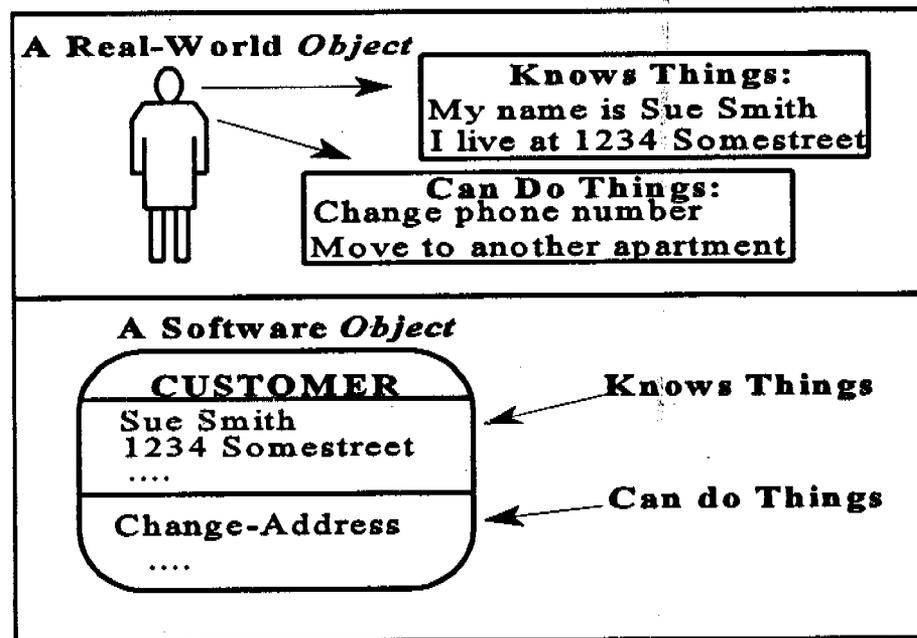


Figure 2.2: A Software Object Models a Real-World Object



## 14 An Introduction to Object COBOL

In OO, the correct term for each specific object is **Instance**. Therefore, we would have, in this example, 3,000 instances of **Customer**. The group of **Customers** is called a **Class**. Thus, we would have a single **Customer Class** with 3,000 instances. Figure 2.3 illustrates **Class** and **Instance**.

<b>Attributes</b>	<b>Methods</b>
Name	
Address	Change-Address
Phone-Number	Change-Phone-Number
Social-Security-Number	

In an Object system, an object's methods are executed when it receives a **Message**, telling it, or more appropriately, asking it to **invoke** a particular method. Thus, if we want to change an address for a particular customer instance, we simply send that object a message requesting it to change the address to a new value. Such a message could appear as:

**CHANGE-ADDRESS, Customer, Sue Smith, 1234 Somestreet**

**CHANGE-ADDRESS** is the **method**, **Customer** is the **class**, **Sue Smith** is the **instance**, and the new address is 1234 Somestreet.

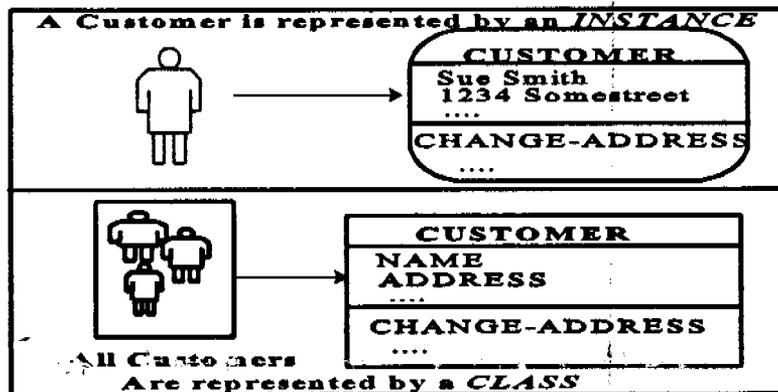


Figure 2.3: Class and Instance

## Class Relationships

Of course, the customer class will not accomplish very much acting alone; it needs to interact with other classes in order to do any processing. While developing the credit union system, we will model the other real-world objects that are needed for the system to be complete. Some of these classes we mentioned earlier are: Account, Checking-Account, and Transaction. Can you name others?



In the real world, these objects interact. Customers make deposits to their accounts, make loan payments, and withdraw funds from savings accounts. Similarly, in our system which models the real-world system, these classes will have *relationships* with each other. For example, a Customer will *have* an Account; Deposits will be *made* to a Checking-Account; Payments will be *made* for a Loan-Account.

In OO, there are three types of relationships between classes: **IS-A**, **CONSISTS-OF**, and **OTHER**. The **IS-A** relationship is the most important of these three. **IS-A** occurs when we have a class that has subclasses which are special types of the class. To illustrate, in our system we have a class called Account. But we actually have three *types* of accounts: Checking-Account, Savings-Account, and Loan-Account. We can then say a Checking-Account **IS-A** Account, a Savings-Account **IS-A** Account, and Loan-Account **IS-A** Account. Another example is the *type* of Loan-Account. We will have a superclass called Loan-Account with subclasses Auto-Loan-Account and Home-Loan-Account. Figure 2.4 depicts this arrangement. Account is the superclass of Checking-Account, Savings-Account, and Loan-Account. Conversely, Checking-Account, Savings-Account, and Loan-Account are all subclasses of Account. Loan-Account is also a superclass of Auto-Loan-Account. Auto-Loan-Account and Home-Loan-Account are subclasses of Loan-Account. A superclass shares its attributes and methods with its subclass. A subclass uses the attributes and methods of its superclass.

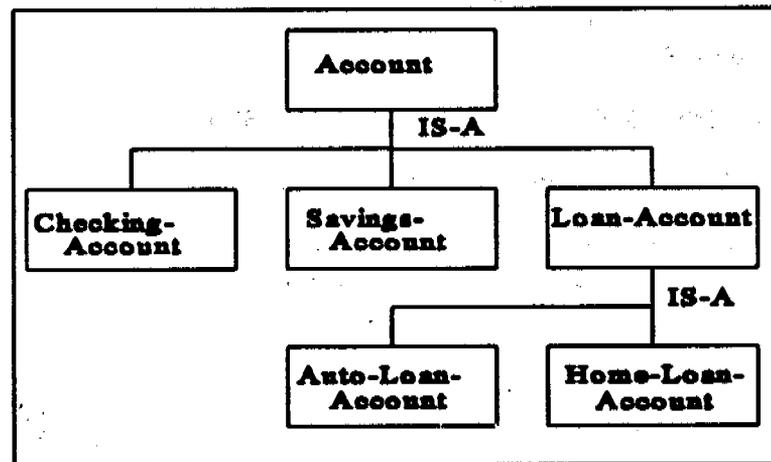


Figure 2.4: IS-A Relationships

Other examples will further illustrate the **IS-A** relationship. Tenured-Professor **IS-A** Professor and a Untenured-Professor **IS-A** Professor. Passenger-Airplane **IS-A** Airplane and Cargo-Airplane **IS-A** Airplane. Graduate-Student **IS-A** Student and Undergraduate-Student **IS-A** Student. These relationships are shown in Figure 2.5.

**CONSISTS-OF** is a whole-part relationship. The whole **CONSISTS-OF** its component parts. For example, an Airplane **CONSISTS-OF** a Fuselage, Wings, Engines, and Landing Gear. A Computer

**Name of Candidate or Committee:** Lake County Republican Central Committee  
**For the reporting period ending:** December 31, 2001

### Summary Page

This summary sheet will give a brief outline of all campaign finance activity during this reporting period.

1.	Amount on hand, if any, at the beginning of the reporting period:	<u>\$272.15</u>
2.	Receipts	
	Schedule A - Direct Contributions	\$305.00
	Schedule B - Fund-Raising Events	\$589.49
	Schedule C - In Kind Contributions	\$0.00
	Schedule D - Other Income	\$0.00
	Total of all Receipts	\$894.49
3.	Total Monetary Receipts	\$894.49
4.	Candidate's Personal Contribution to Own Campaign	<u>\$0.00</u>
5.	Monetary Loans to Candidate or Committee During Reporting Period	<u>\$0.00</u>
6.	Monetary Loans Repaid During Reporting Period	<u>\$0.00</u>
7.	Expenditures - Schedule E	\$27.40
8.	Unpaid Obligations - Schedule F	\$0.00
9.	Amount on hand at the close of this reporting period. *	\$1,139.24

\*The amount on hand at the close of the reporting period should equal the amount of money which the committee has on hand in all checking, savings and cash accounts on last day of the reporting period.

<b>Module</b>	<b>Data Set Name</b>
<b>What To Test</b>	<b>How To Test It</b>
<b>Data Set Information</b>	
<b>Expected Results</b>	<b>Actual Results</b>