1992

Program Maintenance and Code Reuse: Object Oriented Versus Procedure Oriented Programming

Beiramali Moradi

PROGRAM MAINTENANCE AND CODE REUSE
Object Oriented Versus Procedure Oriented Programming

Beiramali Moradi,
B.S. Computer Science

An Abstract Presented to the Faculty of the Graduate School
of Lindenwood College in Partial Fulfillment of the
Requirements for the Degree of
Master of Science
Management Information Systems

1992

ABSTRACT

The purpose of this thesis is to address the problem of the increasing cost of program maintenance in the data processing world. On average, programmers today spend 60% of their time in program maintenance and that figure is increasing at an estimated rate of 1% each year. The cost is staggering and almost every business would like to find ways of reducing maintenance costs. One way to help reduce maintenance and development costs is to improve code reusability. Both maintenance programming and code reusability are the focus of this thesis.

Today's most commonly used programming language style is procedure oriented programming (POP). At nearly every university in the United States, POP is a required course for both scientific and business computer related degrees. Most businesses use POP as the chosen programming language style. Those data processing shops trying to reduce maintenance costs are usually trying to improve techniques used in POP rather than researching new language styles such as object oriented programming (OOP).

OOP requires fewer function names to remember and coordinate than POP requires. A POP language like COBOL, treats its functions as paragraphs or callable sub-programs and each must have a unique name. OOP allows method names to be reused from one class to another and even within the same class. The system takes care of determining which

method should be used. Therefore, OOP is much better at providing fewer functions to remember and coordinate since the system does much of that work for the programmer.

POP languages such as COBOL, have no features or techniques to help prevent **accidental modifications**. OOP helps prevent them by encapsulating classes to limit access to data and methods. Accidental modifications are not totally eliminated with OOP but they can be significantly reduced which helps reduce maintenance costs.

Error detection is difficult in COBOL since it allows all procedures within the program to access any of the data in the program. In OOP, a language feature called encapsulation can be applied to a class to help detect errors. Programmers can spend a lot of time trying to detect errors in a program. Therefore, OOP is able to reduce maintenance costs by helping the programmer to quickly find the errors.

The POP language of COBOL has no specific language feature to help with **change control**. Encapsulation, the same OOP features that helps detect errors can also help limit the impact of a change to a program. Therefore, OOP is better at reducing the amount of time spent in making maintenance changes.

POP languages allow for code reuse by providing the ability to copy source code from a library into a program or call an external sub-program. The programmer must reuse code from the entire program and cannot selectively choose the parts needed or modify the reusable code for only the new program. In OOP, the programmer can pick and choose which data and methods to reuse from existing classes. This ability makes OOP a better language style for **writing reusable code.**

In OOP, through the use of inheritence, **making minor changes to reusable code** is quick and easy. The programmer can create a new class and define only the parts that make the new class different from the existing class. The OOP programmer takes advantage of the code already written without duplicating the code. Code reuse is not possible in POP without duplicating the code and making the change to the new copy. POP, unlike OOP, does not inherently allow for code reuse .

My research into the features and techniques for both POP and OOP supports the hypothesis that, OOP is better than POP at reducing the time spent in program maintenance and leads to improvements in code reusability.

PROGRAM MAINTENANCE AND CODE REUSE
Object Oriented Versus Procedure Oriented Programming

Beiramali Moradi,
B.S. Computer Science

A Culminating Project Presented to the Faculty of the Graduate School
of Lindenwood College in Partial Fulfillment of the
Requirements for the Degree of
Master of Science
Management Information Systems

1992

COMMITTEE IN CHARGE OF CANDIDACY:

Associate Professor Mira Ezvan,
Chairperson and Advisor

Associate Professor Jim Factor,
*Acting Chairperson and Advisor

Adjunct Professor Bob Chant,
Reviewer

Associate Professor Oliver L. Hagan,
Dean of Management Division

* Dr. Jim Factor was the acting Chairperson and Advisor during Dr. Mira Ezvan's absence in the Fall quarter 1991.

ACKNOWLEDGMENTS

Dr. Jim Factor's detailed and thoughtful reviews
of the many drafts of my thesis are much appreciated.

I would also like to thank Dr. Mira Ezvan and Bob Chant
for their suggestions and review of my thesis.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Because of my experience as a maintenance programmer using procedure oriented programming (POP) and because of my interest in object oriented programming (OOP), I have chosen a comparison of these two styles of programming. In particular, I am comparing how their programming techniques and language features help in program maintenance and code reuse. Today's programmers are spending over half their time on maintenance problems. This adds up to a significant cost for all programming shops. Therefore, the hypothesis for this thesis is that OOP is more successful at reducing maintenance costs and improving code reuse than POP. My research will include books and trade magazines where maintenance and code reuse are discussed. I will be gathering current information from several companies[1] to find out about their experiences. My real world experience and academic training will help in this research. To explain some of the points made in this thesis, I will be using COBOL and C++ examples. COBOL will help explain POP techniques and features and C++ will help explain OOP techniques and features.

OOP is considered to be a superior language style to POP in many ways. OOPs breakthrough is that its technology allows the programmer

[1]These companies will include McDonnell Douglas, AT&T, Tripos Assoc., Computer Artisans, Home Savings of America, JWP Controls.

to build large programs from several smaller prefabricated programs.[2] This code reuse requires less programming time as compared to developing similar large programs using POP. Reusing code is rare in POP since the language does not inherently allow it. In OOP, by using smaller simpler programming components, the entire application is usually easier to maintain. Since the code from existing applications can be easily reused, the programs will contain fewer bugs.[3] These ideas represent an introductory comparison of the two language styles. Futher comparions can be found throughout this paper.

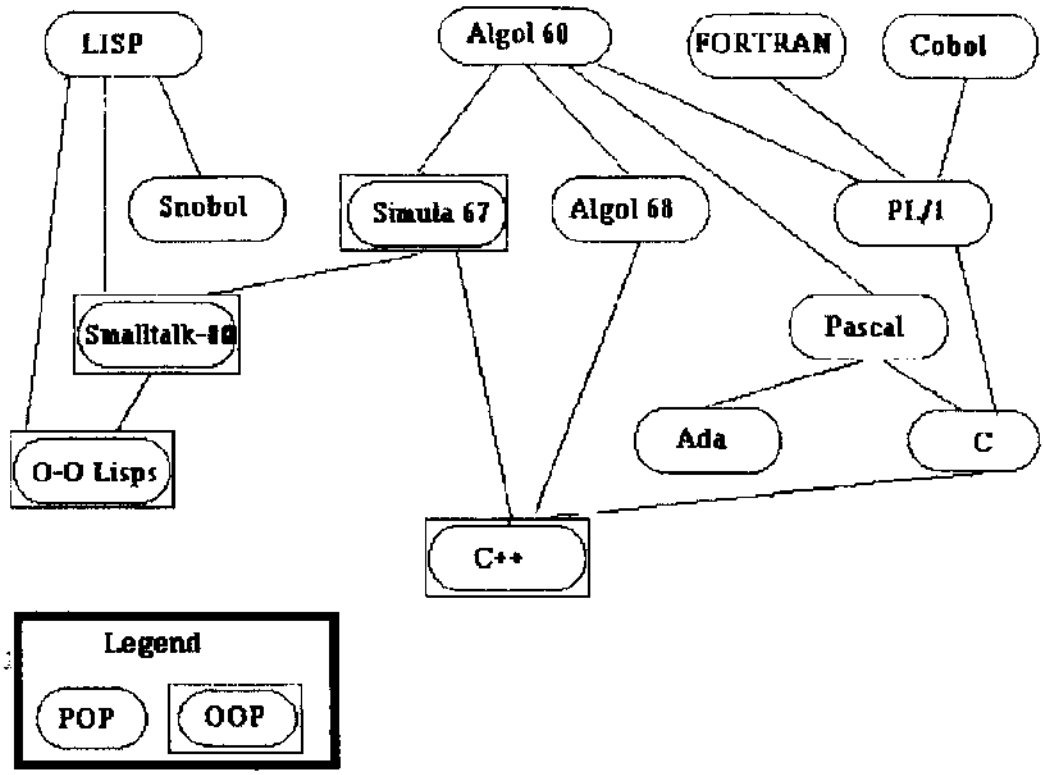
POP dates back to the 1960's with the development of FORTRAN, followed by COBOL and ALGOL-60. FORTRAN and ALGOL-60 were used for scientific programming applications. COBOL was used for business programming applications. Many major programming languages were developed from these three early procedural languages.

OOP was first introduced during the late 1960's with the development of the programming language SIMULA-67. SIMULA-67 evolved from the procedural programming language of ALGOL-60 and contained the new concept of classes and data abstraction. Classes and data abstraction will be explained in chapter 2.

---

[2]James Martin, "OOP Holds Promise of Simplifying Computer Programming", PC Week, 4 September 1989, page 62.
[3]John W. Verity and Evan I. Schwartz, "Software Made Simple", Business Week, 30 September 1991, page 94.

FIGURE 1.0 - Lineage of Major Programming Languages[4]



Nearly all programmers are familiar with POP since it is the traditional style of programming. This style of programming is normally a requirement when earning a computer related degree in college and is more widely used in industry. OOP is normally only an elective class if it is taught at all and is only beginning to be used in industry. For these

[4]Keith E. Gorlen, Sanford Orlow and Perry Plexico, Data Abstraction and Object-Oriented Programming in C++, John Wiley & Sons, 1990, page 4.

reasons I will begin my thesis by explaining the key terms used in OOP. For programmers working with procedure oriented languages, there is more than just syntax to learn.[5] There are many new features to understand and it is these features that make Object Oriented languages stand out from other languages. Some of the features are as follows:

(1) abstract data types
(2) fundamental data types
(3) classes
(4) methods
(5) constructor
(6) objects
(7) messages
(8) inheritance
(9) derived class or sub-class
(10) programming by difference
(11) information hiding
(12) encapsulation
(13) polymorphism
(14) function overloading
(15) templates

I will review in chapter 2, these features and provide an explanation of them and why they are useful.[6] I will also explain the major differences between OOP and POP.

In chapter 3, I will cover an overview of maintenance programming. Maintenance programming includes any type of work

---

[5]Jerry D. Smith, Reusability & Software Construction: C and C++, John Wiley & Sons, Inc., 1990, page xiv.

[6]These features are explained based on many of the definitions and examples provided in the book written by Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley Publishing Company, 1991.

done on a program after it is used in a production mode rather than in test mode. This could mean correcting a programming error, enhancing the code to agree with the users' latest requirements or improving the logic to speed up the run time. Maintenance programming issues discussed will include the following : What are the common reasons for changing production code? Who is making changes to the production code? Why is it considered boring work? What programming tools can help? What techniques and features can help?

Procedural language maintenance programming issues will be discussed in chapter 4. Some of the most common programming languages used today are procedural languages such as COBOL, Pascal, FORTRAN and C. How can POP reduce the number of unique function names? Is there a way to help prevent accidental modifications? If a modification resulted in an error, does POP allow the programmer to easily detect the error? If a data structure changes, how can it impact all the programs that use the structure?

Chapter 5 will discuss the maintenance programming issues related to object oriented languages and such topics as : How does polymorphism help in reducing the number of functions to remember and coordinate? How does encapsulation help in preventing accidental modifications, error detection and change control?

One way of improving maintenance time is to reuse code. In Chapter 6, I will present an overview of code reuse. Reusable code

includes everything from storing program logic for modification and reuse to calling compiled subroutines. Why hasn't code reuse been very successful? How will code reuse help reduce development time and maintenance costs? What options are available to help promote code reuse? What programming tools are available to help in code reuse? What techniques and language features can help with code reuse?

Chapter 7 explains the issues related directly to code reuse with procedural languages and how to address (1) writing reusable code and (2) making minor changes to reusable code. What techniques are available in POP that directly helps in writing reusable code? If a change must be made to reusable code what techniques can be used to make the task easier?

Chapter 8 explains the features of code reuse in OOP for the following areas : (1) writing reusable code and (2) making minor changes to reusable code. Unlike POP languages, OOP languages have many language specific features to help in code reuse. Some of these features include inheritance and templates. What are the benefits of these features?

Chapter 9 will discuss POP and OOP and how they compare. Is POP or OOP better in reducing the number of function names to remember and coordinate? Is POP or OOP better at preventing errors and detecting errors? Is POP or OOP better at controlling the impact of change to a data structure? Chapter 9 will also summarize and compare

OOP and POP to help us determine if OOP makes code reuse easier than POP. If a programmer has to modify reusable code which style makes the job easier? Finally there will be a recommendation for those programming shops using POP and experiencing high maintenance and development costs.

In support of OOP languages, object oriented database management systems (OODBMS) were developed. Although OOP languages can use traditional DBMS based on the relational, hierarchical or network data models, the traditional DBMSs do not support the following[7] :

(1) Supporting complex operations on complex objects
(2) Supporting persistent sharable objects.

Many of the same features that are used in OOP languages have been applied to OODBMS. These features include :

(1) Classes
(2) Objects
(3) Inheritance
(4) Encapsulation

A class can be stored in an OODBMS along with the objects associated with each class.[8] OODBMS objects can be either persistent or transient. In persistent objects, complex data structures can be saved to disk and can be shared from program to program without the programmer having to code the task of translating the information in the

---

[7] Keith Marrs, "Object-Oriented Database Management Systems: The State of the Art", McDonnell Douglas Corporation Report B1659, 1989.
[8] Elisa Bertino and Lorenzo Martino, "Object-Oriented Database Management Systems : Concepts and Issues", Computer, April 1991, page 33.

object between the data base and the program. Each of the classes are encapsulated and the data and methods that are inherited between each of the classes is stored in the OODBMS.

OODBMS directly support OOP to help with program development, maintenance and code reuse. It is estimated that OODBMS usage can save 20% to 30% of development costs above those saved by using an OOP language.[9] Experts in the industry predict that costs associated with the developement and maintenance of a large OOP application using an OODBMS will cost 1/5 to 1/10 of the costs associated with a similar application using traditional methodologies.[10] Unfortunately very few programmers are actually using OODBMS.[11]

A quote from an article printed in the magazine Release 1.0, supports the opinion that OODBMS improves code reuse.[12]

> Object-oriented databases represent the ultimate in reuse and sharing of code. The complex interrelationships and interactions of objects and methods need to be represented only once ... and are then maintained for all users and applications. Application programmers avoid redundant effort.

---

9 & 10 Thomas Atwood, "Applying the Object Paradigm to Databases", Computer Language, September 1990, P. 36 and 41.

11 Three of the programmers that I spoke to have used an OODBMS. These programmers were from McDonnell Douglas, AT&T and Computer Artisians. A second object oriented programmer from McDonnell Douglas, uses relational databases rather than OODBMS.

12 "Objects at Large", Release 1.0, September 19, 1990, P. 4.

This thesis will not address any further the subject of OODBMS. However, I did want to introduce the idea and mention that OODBMS do support improvements in program maintenance and code reuse.

CHAPTER 2

WHAT IS OBJECT ORIENTED PROGRAMMING?

Today the most commonly used style of programming is procedure oriented programming (POP) where languages such as COBOL, Pascal, FORTRAN and C are widely used. Most programmers are *experienced* with POP, some are *familiar* with the concepts of object oriented programming (OOP) and even fewer have any *experience* with OOP.[13] Going from a POP background into the world of OOP can be very confusing. Most procedure oriented programmers have difficulty in changing their mindset to program in an object oriented language.

OOP is a programming style that is very different from POP. OOP was designed with the idea that problem solving should relate directly to the problem itself. Each object included in the program should be able to relate to an item in the problem domain. The communication between these items in the problem domain should relate to the messages that can be sent to each object in the program. For a programming language to be considered object oriented it must allow for classes, objects and inheritance. The most important features of OOP will be explained below.

---

[13] Every programmer I spoke to knew a POP language, but it was difficult to find programmers that knew OOP. There are usually one or two departments in very large companies or a specialized small company that use OOP. Of the companies that I spoke to AT&T, McDonnell Douglas and Computer Artisians use OOP.

The centerpiece of OOP is the **abstract data type**.[14] Data types are either fundamental data types or abstract data types. **Fundamental data types** include integer, real, character, and floating decimal.[15] The language decides what data values are valid for each of the fundamental data types and decides what operations can be performed on them. Abstract data types are just like fundamental data types except the programmer rather than the language decides the data that will be valid and what type of operations can be performed on that data. For example, if a program needs to generate a report based on employee information, the employee could be defined as an abstract data type including such operations as a request to determine if the employee is a member of a union or to print the information related to an employee.

The way to define an abstract data type is by defining a **class**. FIGURE 2.0 shows the basic format in C++ for defining the class **Employee**. The class definition for the abstract data type of Employee should contain the data and operations relating directly to employee. The data and operations are directly related and they cannot be separated.[16] The operation ***Employee*** creates a new instance of the class **Employee**. The operation ***IsUnion*** determines if an employee is a member of a

---

[14] Richard S. Wiener and Lewis J. Pinson, An Introduction to Object-Oriented Programming and C++, Addison-Wesley Publishing Company, 1988, page 1.

[15] Terrence W. Pratt, Programming Languages Design and Implementation, Prentice Hall, Inc., 1984, page 44.

[16] Jeffrey Duntemann, "OOP : A New Perspective on Code and Data", PC Week, 14 November 1988, page 69.

union. The operation ***Print***, prints information related to that employee. The commands necessary to execute the operation are found within the symbols { } immediately following the operation name. In C++, comments that are on a line by themself are found between the symbols /* and */. Comments can also be written on the same line as code as long as the comment is preceeded by the symbols //. The Private and Public sections of a class will be described later. The basic class Employee format is as follows:

FIGURE 2.0 - Basic format for defining a class in C++

```
class Employee

private :
/*  Lists data that can only be accessed by this class  */

public :
/*  Lists operations that can be accessed by any class or program  */
   Employee   // operation 1 - creates a new employee
   { . . . }      // commands to perform Employee operation
   IsUnion    // operation 2 - determines if employee is in union
   { . . . }      // commands to perform IsUnion operation
   Print      // operation 3 - prints information for to employee
   { . . . }      // commands to perform Print operation
```

A class is a way of describing similar "things". A class can be defined by a programmer or taken from a library of classes. These classes can later be referenced by other classes or directly by a program.

In the class definition, the programmer must define the type of data and operations that are valid for the instances of that class. The data for

class **Employee** can be defined in the private section of the class definition as follows:

```
long soc_sec_no;      // Long int. number for social security number
```

Long is a fundamental type of integer. Integers can either be defined as long or short depending on how large the greatest possible value will be. For social security number, the largest number will be 9 digits of all 9's. For processing on a microcomputer, the number 999999999 exceeds the largest possible value of 32,767 for short integers, so social security number must be declared as a long integer.

Another word for an operation within a class is a **method**. In C++ the constructor method must have the same name as the class. The **constructor** method creates a new instance of a class. The constructor method ***Employee*** in FIGURE 2.1 requires 4 parameters be passed to create a new employee. The first parameter of new_ln will contain a character string value like "Doe" that will be passed to LastName and later set equal to last_name. The second parameter of new_fn will contain a character string value like "Jane" that will be passed to FirstName and later set equal to first_name. The third parameter of new_jcd will contain a character string value like "52100" that will be passed to JobCode and later set equal to job_code. The fourth parameter of new_ssn will contain a long integer value like 111223333 that will be passed to ssn and later set equal to soc_sec_no. The constructor method

definition from FIGURE 2.1 is defined as follows :

```
Employee (char LastName[20], char FirstName[15],
          char JobCode[5], long ssn)
  { . . . }
```

In FIGURE 2.1, both the method ***IsUnion*** and the data variable in_union are defined as boolean. A **variable** is a field assigned a name, a set of attributes, a reference and a value.[17] A **boolean** is an identifier that contains either true or false. In the method ***IsUnion***, if the employee is found to belong to a union, the boolean variable in_union is set to true. To determine the value of in_union outside of the class **Employee**, the method ***IsUnion*** must be used. The string function **strncmp**[18] compares the first *n* characters of two strings. Where *n* is an integer. The function returns a value of zero if the two strings are equal. The commands for a method are between the symbols { and } and follow the method name.

---

[17] Ellis Horowitz, Fundamentals of Programming Languages, Computer Science Press, 1983, page 82.

[18] Borland C++, Library Reference, Borland International, 1991, page 499.

The method ***IsUnion*** from class **Employee** is defined as follows :

```
/*  Returns true if the employee is union otherwise returns false  */
boolean IsUnion ()
 {
if (strncmp (family, "100",3) ) == 0)
   in_union = true;
else
   {
   if (strncmp (family, "200",3) ) == 0)
        in_union = true;
   else
       in_union = false;
   };
    return in_union;
```

In FIGURE 2.1, void is used with the method ***Print***. Void is found before a method's name when the method does not return a value to the program. The **cout** command and the insertion symbols << in the method ***Print*** , causes the variables or literals following the command to be printed as output.

```
/*  Prints 3 strings and 1 long decimal  */
/*  which are last name, first name and social security number*/
void Print()
{
  cout << last_name << " "<< first_name << " "
         << soc_sec_no << " " << job_code << " ";
}
```

Data and methods can be defined as private, protected or public. If data and methods are defined as **private**, they can be used only by the defining class. If data and methods are defined as **protected**, they can be used by the defining class and by any sub-class of the defining class. If data and methods are defined as **public**, they can be used by the defining

class and any other class. FIGURE 2.1 shows in detail three valid methods in the public part of the class **Employee**. The first method, called ***Employee***, tells what is necessary to build or construct an instance of class **Employee**. This constructor requires the last name, first name, job code and social security number of the employee to be passed as parameters. Rather than using the arrays that were passed as parameters, C++ requires that a pointer to the array be used. Therefore pointers were defined for last name, first name and job code. The **job code** field defines for the employee, the skill level and type of work. The **family** field defines the employee's type of work. The second method, named ***IsUnion*** was described above. The third method, called ***Print***, will print a line of information related to the employee.

FIGURE 2.1 - **Class Employee in C++**

```
#include <iostream.h>
#include <string.h>
/* Beginning of Class Employee definition */
enum boolean {false, true}; // false = 0 ; true = 1

class Employee {              // Class Employee definition
private:                      // Accessible only to this class
 char *last_name;             // pointer to last name
 char *first_name;            // pointer to first name
 char *job_code;              // pointer to job code
 char grade[2];               // 2 character string for the job grade
 char family[3];              // 3 character string for job family
 long soc_sec_no;             // Long int. value for social security number
 boolean in_union;            // Boolean that tells if employee is in a union
```

FIGURE 2.1 continued

```
public:                          // methods accessible by all classes
 /* If 3 string and 1 numeric parameter is passed to the Employee   */
 /* constructor, the private fields will be set equal to  values passed */
 Employee (char LastName[20], char FirstName[15],
           char JobCode[5], long ssn)
 {
 last_name = LastName;     // Copies name to private data item
 first_name = FirstName;   // Copies name parm to private item
 job_code = JobCode;       // Copies job code parm to private item
 soc_sec_no = ssn;         // Copies soc sec no parm to private data item
    /* Job grade is the same as the first 2 pos of job code */
 grade[0] = job_code [0];  // Position 1 of job code to pos 1 of grade
 grade[1] = job_code [1];  // Position 2 of job code to pos 2 of grade
    /* Job family is the same as the last 3 pos of job code */
 family[0] = job_code[2];  // Pos 3 of job code to pos 1 of family
 family[1] = job_code[3];  // Pos 4 of job code to pos 2 of family
 family[2] = job_code[4];  // Pos 5 of job code to pos 3 of family
 }

/* Returns true if the employee is union otherwise returns false */
boolean IsUnion ()
{
 if (strncmp (family, "100",3) ) == 0)
    in_union = true;
 else
{
  if (strncmp (family, "200",3) ) == 0)
      in_union = true;
  else
      in_union = false;
  };
   return in_union;
}

/* Prints 3 strings and 1 long integer                    */
/* which are last name, first name, job code and ssn      */
void Print ()
{
  cout << last_name << " " << first_name << " "
       << soc_sec_no  << " " << job_code << " ";
}
};                  // End of Class Employee Definition
```

Just like you can define a variable to be one of the fundamental data types, you can also define a variable to be of an abstract data type. When a variable is defined to be an abstract data type, the variable is called an **object**. An object is another word for an instance of a class. In the example below from the main program in Figure 2.2, the object **NewEmp** is assigned to the class **Employee**. The command below tells the name for the constructor method ***Employee*** that must be executed to create the new object **NewEmp** along with the data parameters.

Employee **NewEmp**(new_ln, new_fn, new_jcd, new_ssn);

From FIGURE 2.2, the employee's information should only be printed on the report if the employee is part of a union. Since the boolean variable in_union is in the private area of the class **Employee**, the program cannot directly access the variable in_union. The program must use the method ***IsUnion***, to determine whether to print the employee's information.

```
/*  only print information on union employees  */
uniontest = NewEmp. IsUnion ();
if (uniontest == true)
   NewEmp. Print ();
```

To cause a method to begin executing, a **message** must be sent to the object that contains the method. This message must contain the name of the method to be executed and any data that must be passed as

parameters to the object. There are three types of messages: [19]

(1) Requests for data from within the object.
(2) Requests for the object to accept new data.
(3) Requests for object to perform special operations.

In the following example from the main program in FIGURE 2.2, **NewEmp** is the object name for a definition of class **Employee**.

NewEmp.Print ();

Print is the message name that relates directly to the method ***Print*** in the class **Employee**. In the body of the program, the object name and the message should be separated by a period. Nothing appears within the parenthesis if no parameters are passed to the method. At execution time the message ***Print*** is sent to the object ***NewEmp***. This message causes the method ***Print*** in the class **Employee** to bind the formal and actual parameters from object **NewEmp** and print the information from the object **NewEmp**.

A simple main program is shown below in FIGURE 2.2 that references the class **Employee** in FIGURE 2.1 and generates a report of only union employees.

---

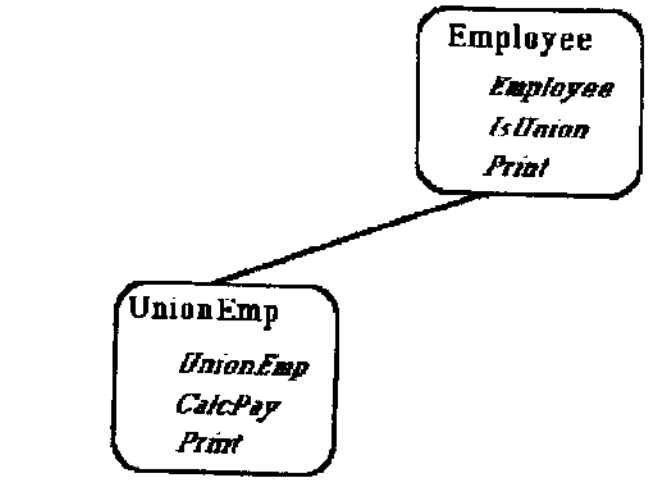
[19]Mark Mullin, Object Oriented Program Design with Examples in C++, Addison-Wesley, 1989, P. 59.

FIGURE 2.2 - Main Program

```
main () {      // Start of main program logic in C++

 char new_ln[20], new_fn[15], new_jcd[5];
 double new_sl, new_hr;
 long new_ssn;
 boolean uniontest;
 do       // start of loop through employee database
 {
 /*  This is where the command to read the database should go.        */
 /*  The database fields for last name is moved to new_ln.            */
 /*  The database fields for first name is moved to new_fn.           */
 /*  The database fields for job code is moved to new_jcd.            */
 /*  The database fields for social security numbers is moved to new_ssn.*/

 /*  Construct a NewEmp object from the Employee class.               */
 Employee NewEmp(new_ln, new_fn, new_jcd, new_ssn);
 /*  only print information on union employees .                      */
 uniontest = NewEmp. IsUnion ();
 if ( uniontest  == true)
    NewEmp. Print();
 }
 /*  Stop looping when the database end of file is reached            */
 while { ... code goes here to loop while employee recs remain ... }
}
```

To define a new class that looks almost exactly like an existing class, the programmer can reuse the code in the existing class and use any of the data and methods available to the new class. The ability for a new class to reference data and methods from an exisiting class is called **inheritance**. When one class inherits data and methods from an existing class, the original class is called the root class. The new class is called the **derived class** or **sub-class**. FIGURE 2.3 below shows that class **UnionEmp** is derived from the root class **Employee**.

FIGURE 2.3 - Class **UnionEmp** inheritance from class **Employee**



class UnionEmp inherits methods *Employee, IsUnion and Print* from class Employee

FIGURE 2.4 shows the class definition for the sub-class of **UnionEmp** as derived from the root class **Employee.**

***FIGURE 2.4* - Sub-class of UnionEmp**

```
class UnionEmp : Employee {    // Class UnionEmp definition
private:                       // Accessible only to this class
 double salary;                // holds salary for a union Employee
 double hours;                 // holds hours worked for a union employee
 double tot_hours;             // holds total hours
 double otpay;                 // holds the calculated overtime pay amount
 double totpay;                // holds the calculated total pay amount
 double othours;               // holds the calculated overtime hours
 double otrate                 // constant overtime rate
```

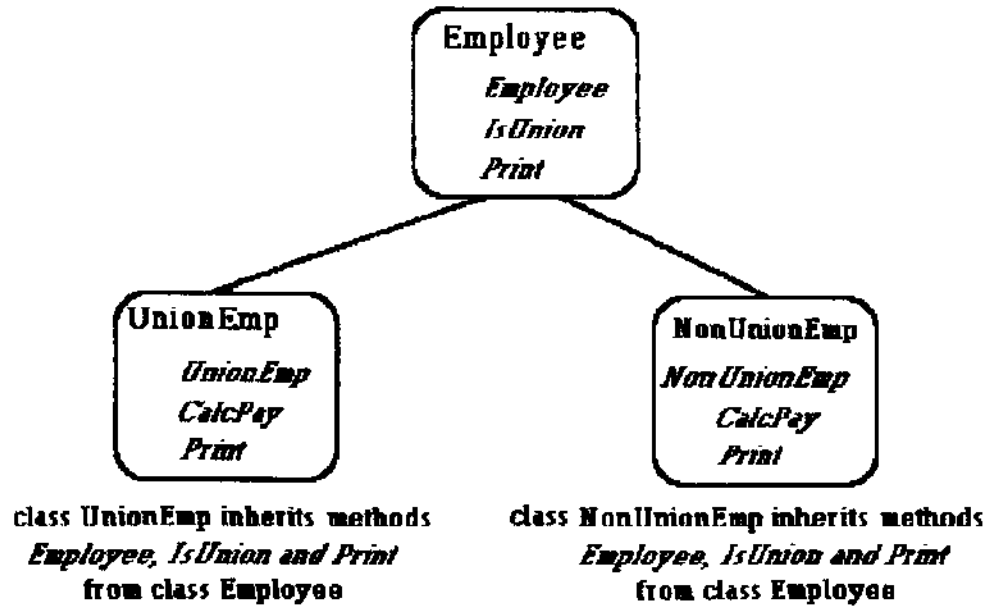
FIGURE 2.4 continued

```
public:                    // methods accessible by all classes
 /*  If 3 string and 3 numeric parameter is passed to the UnionEmp */
 /*  constructor, the private fields will be set equal to the values passed */
 UnionEmp(char LastName[20], char FirstName[15],
                char jobcode[5], long ssn, double sal, double hrs) :
          Employee (LastName, FirstName, JobCode, ssn);
      {   otrate = 1.5;
          salary = sal;
          hours = hrs;
          tot_hours = hours;
      }
 void CalcPay()
      {
      othours = hours - 40;
      IF (othours < 0 )
          othours = 0;
      else
          hours = 40;
      otpay = (othours * otrate * salary);
      totpay = (hours * salary) + otpay;
      }
 void Print()
      {
      Employee :: Print();
      cout << salary << " " << tot_hours << " " << totpay << " "
           << otpay << "\n";
      }
}                    // End of class UnionEmp definition
```

In general, the data and methods that a class inherits are all those defined in the public and protected areas of the root class. In our example there was no protected area, so the sub-class **UnionEmp** inherits only the public methods ***Employee***, ***IsUnion*** and ***Print*** from class **Employee**. The programmer must mention only the data and methods that are needed in addition to those that are inherited. Since the programmer of the derived class only has to mention the data and methods that are different from those available in the root class, this

feature is called programming by difference. Class UnionEmp must have its own constructor method *CalcPay* that can calculate the weeks pay for a union employee and a method *Print* to print the detail related only to union employees. The data fields in class **UnionEmp** are needed only by the derived class and are kept in the private area of the class definition. These data fields are used to calculate payroll dollars. A sub-class was set up for union employees since it has characteristics that make this type of employee different from other types of employees. One characteristic is a unique formula to calculate payroll dollars. Chapter 5 will build on this example and also shows a class for non-union employees. FIGURE 2.5 illustrates that class **UnionEmp** and class **NonUnionEmp** are derived from the root class **Employee.**

FIGURE 2.5 - Class **UnionEmp** and **NonUnionEmp** inheritance from class Employee



As stated earlier, objects are of a certain type of class. Those objects contain methods that manipulate the data. When a program passes a message to cause an action, the program has no information of how the action was completed. To the program, the object acts like a "black box". All the program knows is that information is passed to the object and a result happens. This process is called **information hiding** since the information within the object is hidden from other objects. There are 2 main benefits to information hiding as noted below by Stanley Lippman.[20]

(1) **Change Control** - If a data representation in a class changes, only the members of the class would have to be modified. User programs would not need to be modified.

(2) **Error Detection** - If an error occurs in the manipulation of a class data member, only the class member functions need to be evaluated to find the source of the problem.

This method of hiding information and only allowing certain types of actions (i.e. methods) to be performed against the data is called **encapsulation**. Encapsulation acts like a filter that controls how an object communicates with other objects and programs. From FIGURE 2.2, the command shown below constructs a new object **NewEmp**.

Employee NewEmp (new_ln, new_fn, new_jcd, new_ssn);

---

[20]Stanley Lippman, C++ Primer, 2nd Edition, pg 51.

Object **NewEmp** is considered to be an encapsulated object since the data items in the private section of the object **NewEmp** can only be accessed by the methods within object **NewEmp**. The methods in the public areas of object **NewEmp** allow limited access by other classes and programs. The type of access given to classes and programs outside of object **NewEmp** is controlled by object **NewEmp**. For example, a program can determine if the employee is a union member by sending the message ***IsUnion*** to the object **NewEmp**. The programmer cannot change the value of the boolean **in_union**, a variable in the private section of class **Employee**, since it is accessible only to object **NewEmp**. The value of boolean **in_union** is only accessible through the method ***IsUnion***. The value of the **otrate** (1.5) is not accessible at all by any class or program outside of class **UnionEmp**. **Otrate** is not even available through a method. Encapsulating specific data items helps to prevent accidental modifications. If there should never be a reason for this value to be modified directly outside the class definition then the data should be made private and therefore encapsulated from misuse.

As mentioned earlier, both the class **Employee** and the class **UnionEmp** will have a method ***Print***. The same name can be used to execute both the method ***Print*** for the class **Employee** and the method ***Print*** for class **UnionEmp**. The system takes care of deciding which method to execute based on the type of object the message is sent to. The system determines which method to execute at run time. The ability to resolve which set of procedures to execute is called **polymorphism**.

What follows is a command found in FIGURE 2.2 that issues the message *Print*.

NewEmp. ***Print*** **();**

Since object **NewEmp** is derived from the class **Employee**, the system knows to execute the method ***Print*** found in the class **Employee** rather than the method ***Print*** found in the class **UnionEmp**. Similarly, for an object **NewUnionEmp**, the system would use the method ***Print*** found in the class **UnionEmp** in the following command.

NewUnionEmp. ***Print*** **();**

With the system resolving which method to execute, less work is required of the programmer. This certainly reduces the number of messages that a programmer must remember when using different classes and decreases the chances of errors.

A capability exists that allows multiple functions to have the same function name within one class. This capability is useful because there will be **fewer functions names to remember and coordinate.** The system takes care of executing the correct function based on the type or number of data items being passed as parameters. The only requirement of these same-name-functions is that there be a difference in either the number of parameters in each function or that the parameters types be

different. This ability of the system to have several functions with the same name within one class is called function overloading.

Templates are a new feature to some object oriented languages. Templates can be used when multiple functions contain the same code except they process different types of data and return different types of data. If a programmer needed a function to return the minimum integer value found in an array of integers, the function would be as found in FIGURE 2.6 below.

FIGURE 2.6 - Function for finding minimum integer in array

```
int min ( int* array, int size) {
int min_val = array[0];
for (int ix = 1; ix < size; ++ix)
    if (array[ix] < min_val) min_val = array[ix];
return min_val;
}
```

Now lets assume that the same programmer also needed a function to return the minimum real value found in an array of real numbers. FIGURE 2.7 is an example of how that function may look.

FIGURE 2.7 -
Function for finding the minimum double precision number in an array

```
double min ( double* array, int size) {
double min_val = array[0];
for (int ix = 1; ix < size; ++ix)
    if (array[ix] < min_val) min_val = array[ix];
return min_val;
}
```

Both functions perform the same task of searching through an array of numbers to find the minimum number. The only difference is that in the first function the array and returned number are of type integer and in the second function the array and returned number are of type real. Instead of coding two separate functions, one function could be written as a template function with the data type as a variable. Templates require that a skeleton for the function be written with an argument passed to the function telling what data type will be processed. FIGURE 2.8 shows a function template where the argument defining the data type is *TYPE*.

FIGURE 2.8 - Function for finding minimum **unknown type** in array

```
template <class TYPE >
TYPE min (TYPE * array, int size) {
 TYPE min_val = array[0];
 for (int ix = 1; ix < size; ++ix)
      if (array[ix] < min_val) min_val = array[ix];
 return min_val;
}
```

To process this template against an array of integer numbers and return an integer number, the command must be issued as follows where int_array is an array filled with integer numbers :

```
int int_result = min(int_array, size);
```

To process this template against an array of real numbers and return a real number, the command must be issued as follows where double_array is an array filled with real numbers :

```
double doub_result = min(double_array, size);
```

Templates help in program maintenance since there will be fewer functions to maintain but templates can significantly improve code reusability.

SUMMARY

The major differences between POP and OOP are the capabilities of data abstraction, encapsulation, inheritance and polymorphism.[21] POP languages like COBOL for example, allow only the fundamental data types of characters, integers, binary, and single-precision floating decimal but do not allow for abstract data types. POP languages also do not allow encapsulation, inheritance and polymorphism. These features help to make OOP a more flexible language than POP. Chapters 3, 4, and 5, will support the suggestion that object oriented programs are easier to maintain than procedure oriented programs.

---

[21]Ray Duncan, "Power Programming, Redefining the Programming Paradigm: The Move Toward OOPLs", PC Magazine, 13 November 1990, page 526.
Zack Urlocker, "Teaching object-oriented programming", Journal of Object-Oriented Programming, July-August 1989, page 45.

CHAPTER 3

MAINTENANCE PROGRAMMING OVERVIEW

On average programmers today spend 60% of their time maintaining programs and this is increasing at approximately 1% each year.[22] It is also estimated that 70% of the investment on an application over its entire life is spent on maintenance.[23] Investigation has shown that one of the major causes of new development projects not meeting scheduled implementations is because of unplanned maintenance.[24] Maintenance programming includes any change made to a program once it is in a production mode no matter what the reason for the change. Common reasons for change are :[25]

(1) Correct a "bug" in the program.
(2) Improve the system as requested by the user.
(3) Improve logic to speed up the application.

Even changes that seem to be minor, can take a long time to complete. There are many reasons that cause program maintenance to be so time consuming. The cause is usually that a program is confusing and the person doing the job has trouble determining the best place to make the

---

[22]Martin Butler and Robin Bloor, "Object Orientation," DBMS, July 1991, page 17.
[23]Diane Drotos and Stella Skerlec, "Creating a Rewarding Maintenance Environment," Computing Canada, 25 October 1990, page 42.
[24]Michiel Van Genuchten, "Why is software late?," IEEE Transactions on Software Engineering, July 1991, page 582.
[25]Lowell Jay Arthur, Software Evolution : The Software Maintenance Challenge, Wiley-Interscience Publication, 1988, pages 5-6.

change. The programmer must make sure that the change being made to the program does not cause new problems.

The person making the change is usually not the same person that wrote the program in the first place. Usually maintenance programmers are the newest programmers in a department. The more experienced programmers are usually working on the development of new applications. Since changes are being made by inexperienced programmers, they often have trouble finding the best way to make a change to a program. These changes often act like bandage fixes which can turn an already difficult to maintain program into a nightmare.

Maintenance is demotivating work for most programmers and new development is considered motivating.[26] When designing and programming an application from scratch, the programmer has a lot of room for creative expression. When doing program maintenance, the flow of the application is complete and the programmer must fit into the structure already set up. Creatively this is not as challenging to most programmers.

It is very frustrating and can be difficult to try and determine how a change to a program will affect the rest of the program. Programs are often structured so that what could have been a simple change becomes a major change. The ability to isolate changes to small sections of code can help reduce maintenance time. In some languages this could mean using

a programming technique and in other languages this is done by built in language features.

Because of the amount of time spent in program maintenance, you can see why companies are trying to improve the process. There are several options available. Some of the options to help reduce program maintenance time are :[26]

(1) Use *programming tools* to improve testing and making the changes.

(2) Make sure programmers use the *language techniques and features* available for the chosen language.

(3) *Choosing a different language or style of language* to find one better able to improve maintenance programming.

Each of these will be considered in the following 3 sections :

(1) Programming Tools
(2) Techniques and Features
(3) Choosing a Different Language

PROGRAMMING TOOLS

Programming tools are available that can help reduce the time spent in program maintenance. For each maintenance task, a tool may be

[26]Girish Parikh, Techniques of Program and System Maintenance, QED Information Sciences, Inc., page 278.

available that can help reduce the time spent doing that task. Three of those tools are as follows :

(a) **Debugging Tools**
(b) **Conversion Tools**
(c) **Cross Reference Listings**

Locating a bug in a program takes a lot of time. **Debugging tools**[27] are available for many languages to help find program bugs. These debuggers let the programmer step through the program line-by-line to see just how a program reacts to the data it processes. Variables can be displayed to show when the value changes. Being able to see what functions are executed and what the value of certain fields are, helps the programmer identify problem areas quickly which reduces testing. Debuggers are available for both POP and OOP.

Trying to read unstructured procedural code can be very time consuming. **Conversion tools** exist that allow code to be read into a conversion program that changes it into structured code. Structured code is an accepted style of programming that helps makes POP code easier to understand and therefore easier to maintain.

Trying to find where variables and procedures are defined and used in a program can be a difficult and time consuming task. Fortunately, many POP compilers (e.g. COBOL, FORTRAN, Pascal, etc.) offer **cross**

[27]Programmers from every company I spoke with used a debugging tool regardless of whether the programmer was using OOP or POP.

reference listings that shows where a variable or procedure is defined and used in a program. These cross reference listings are very helpful when trying to understand the impact that changing a module will have on the rest of the program. The cross reference listing can act as documentation and serves as an important tool in the maintenance process.

TECHNIQUES AND FEATURES

Applying programming techniques to a language can improve the maintenance process. For example one could make the program self documenting by adding comments or using structured programming techniques.[28] Unfortunately techniques must be implemented by the programmer. Nothing forces the programmer to use a technique. Many techniques are meant to help the maintenance programmer at a later date. The development programmer is usually not concerned about the maintenance programmer. Everything in the program makes perfectly good sense to the development programmer so spending the time to apply some of the techniques to help the maintenance programmer seems like wasted time and often these techniques do not get applied.

Using language features are more successful at reducing maintenance costs then using programming techniques. An example of a

---

[28]A programmer from Tripos Associates presented documented programming standards that stressed the use of internal source code documentation.

language feature that helps reduce maintenance time is encapsulation in OOP. Encapsulation reduces the time it takes to find errors and makes it easy to determine the impact of a change to a class. Polymorphism is an OOP language feature that reduces the number of function names to remember and coordinate. How Encapsulation and Polymorphism help the maintenance task is explained in detail in chapter 5.

All languages have some basic programming techniques and features that can be used to help reduce maintenance time. POP languages have fewer such language features than OOP languages.

CHOOSING A DIFFERENT LANGUAGE

If the language chosen by a programming shop is not performing well in the area of maintenance, a review must be done to determine if the selected language is being used to its best or if another language should be chosen. If the program maintenance does not improve by using the tools, techniques and features for a chosen language, then the programmer should consider choosing a different language. There may be an improvement by going from one procedural language to another but there may be an even greater improvement by going from a procedural language to an object oriented language.

Using a language that has features to help a programmer code so that program maintenance is naturally improved is better than using a language where it is up to the programmer to remember to apply special techniques to the code.

SUMMARY

Since POP and OOP have similar programming tools available, these tools will not be discussed in the following chapters. What I will concentrate on in chapters 4 and 5 are the language techniques and features available to help reduce program maintenance time in four specific problem areas found in FIGURE 3.0.

FIGURE 3.0

1. Fewer function names to remember and coordinate.
2. Accidental modifications.
3. Error detection.
4. Change control.

CHAPTER 4

MAINTENANCE PROGRAMMING USING POP

There are tools, techniques and language features that can be applied to procedure oriented programming (POP) languages to help improve program maintenance. This chapter deals with POP, and how together the language style and the programmer are able to address the four program maintenance areas in FIGURE 3.0.

To address these areas, the following example will be referenced throughout this chapter. Assume that a programmer was asked to write a program to calculate the weekly pay for the employees of Company A. Company A uses the POP language COBOL. COBOL was chosen as the language for the following example because the features and problems it has in the area of maintenance are characteristic of other POP languages and COBOL is the most widely used POP language in business.

EXAMPLE

PROBLEM DISCUSSION

Company A has both union and non-union employees. A program must be written that takes each employee's salary and current week's hours to calculate the week's pay and print a detailed report.

Union employee description:

- Paid by the hour and receive 1.5 times their hourly rate for each hour of overtime.
- Salary is stored as an hourly amount.
- Report should contain the following information : (1) name (2) SSN (3) job code (4) salary (5) hours worked (6) total pay (7) overtime pay. Sample output follows :

NAME		SSN	JOBCODE	SALARY	HRS	TOTPAY	OTPAY
Smith	Mary	222334444	52100	$10.00	44	$460.00	$60.00

Non-union employee description:

- Paid a weekly salary and are paid the hourly equivalent of their weekly salary for each hour of overtime.
- Salary is stored as a weekly amount.
- Report should contain the following information for non-union employees (1) name (2) SSN (3) job code (4) salary (5) hours worked (6) total pay. Sample output follows :

NAME		SSN	JOB CODE	SALARY	HRS	TOTPAY
Doe	John	111223333	58444	$450.00	40	$450.00

IMPLEMENTATION

Positions 1-2 of the job code field indicate the skill level or grade of the employee. Positions 3-5 of the job code indicate the family of work the employee belongs to. If the job family is equal to union number 100 or union number 200, the employee is considered a union employee.

FIGURE 4.0 shows pieces of the COBOL program that would perform the necessary calculations and print the report. To reduce the length of the program only the important lines of code will be included in the program. An asterisk before a line of code means that line is a comment.

FIGURE 4.0 - Employee Payroll Report in COBOL

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
FILE-CONTROL.
DATA DIVISION.
FILE SECTION.
  FD  REPORT-FILE
        DATA RECORD IS REPORT-RECORD.
  01 REPORT-RECORD                PIC X(133).
* The data area is in the working-storage section.
WORKING-STORAGE SECTION.
  01 100-EMPLOYEE-RECORD.
     05  100-LAST-NAME            PIC X(20).
     05  100-FIRST-NAME           PIC X(15).
     05  100-DEPARTMENT           PIC X(04).
     05  100-JOB-CODE.
         10  100-GRADE            PIC X(02).
         10  100-FAMILY           PIC X(03).
             88  100-88-UNION-EMPLOYEE          VALUE "100",
                                                      "200".
     05  100-SSN                  PIC 9(09).
     05  100-SALARY               PIC 9(6)V99.
     05  100-HOURS                PIC 9(3)V99.

  01 200-HOLD-AREAS.
     05  200-UNION-OTRATE         PIC 9(3)V99     VALUE 1.5.
     05  200-HOLD-HOURS           PIC 9(3)V99     VALUE 0.
     05  200-OTHOURS              PIC 9(3)V99     VALUE 0.
     05  200-OTPAY                PIC 9(6)V99     VALUE 0.
     05  200-TOTPAY               PIC 9(6)V99     VALUE 0.
     05  200-HRLY-RATE            PIC 9(3)V99     VALUE 0.
     05  200-END-OF-FILE          PIC X(03)       VALUE "NO ".

  01 300-CONSTANT-AREA.
     05  300-FULLTIME-HOURS       PIC 9(03)       VALUE 40.
```

FIGURE 4.0 continued

```
01 400-PRINT-DETAIL-LINE.
   05  FILLER                 PIC X(03).
   05  400-LAST-NAME          PIC X(20).
   05  FILLER                 PIC X(03).
   05  400-FIRST-NAME         PIC X(15).
   05  FILLER                 PIC X(03).
   05  400-SSN                PIC 9(9).
   05  FILLER                 PIC X(02).
   05  400-JOB-CODE           PIC X(05).
   05  FILLER                 PIC X(02).
   05  400-SALARY             PIC $ZZZ,ZZ9.99.
   05  FILLER                 PIC X(03).
   05  400-HOURS              PIC ZZ9.99.
   05  FILLER                 PIC X(03).
   05  400-TOTPAY             PIC $ZZZ,ZZ9.99.
   05  FILLER                 PIC X(03).
   05  400-OTPAY              PIC $ZZZ,ZZZ.ZZ.
   05  FILLER                 PIC X(03).

* The main program and paragraphs are found in the procedure division.
PROCEDURE DIVISION.
0000-MAINLINE.
  PERFORM 1000-INITIALIZATION.
  PERFORM 2000-PROCESS-EMPLOYEE-RECS
     UNTIL 200-END-OF-FILE = "YES".
  PERFORM 3000-TERMINATION.

1000-INITIALIZATION.
*   OPEN FILES, DO INITIAL READ

2000-PROCESS-EMPLOYEE-RECS.
  PERFORM 2100-CALCPAY.
  PERFORM 2200-PRINT-REPORT.
  PERFORM 8000-READ-EMPLOYEE-REC.

2100-CALCPAY.
  IF 100-88-UNION-EMPLOYEE
     PERFORM 2110-CALCPAY-UNION
  ELSE
     PERFORM 2120-CALCPAY-NONUNION.
```

FIGURE 4.0 continued

```
2110-CALCPAY-UNION.
  MOVE 100-HOURS TO 200-HOLD-HOURS.
  COMPUTE 200-OTHOURS =
                200-HOLD-HOURS - 300-FULLTIME-HOURS
  IF 200-OTHOURS < 0
     MOVE ZERO TO 200-OTHOURS
  ELSE
     MOVE 300-FULLTIME-HOURS TO 200-HOLD-HOURS.
  COMPUTE 200-OTPAY =
        (200-OTHOURS * 200-UNION-OTRATE * 100-SALARY).
  COMPUTE 200-TOTPAY =
        (200-HOLD-HOURS * 100-SALARY) + 200-OTPAY.

2120-CALCPAY-NONUNION.
  MOVE 100-HOURS TO 200-HOLD-HOURS.
  COMPUTE 200-OTHOURS =
         200-HOLD-HOURS - 300-FULLTIME-HOURS.
  COMPUTE 200-HRLY-RATE =
        100-SALARY / 300-FULLTIME-HOURS.
  IF 200-OTHOURS < 0
     MOVE ZERO TO 200-OTHOURS
  ELSE
     MOVE 300-FULLTIME-HOURS TO 200-HOLD-HOURS.
  COMPUTE 200-TOTPAY =
     (100-SALARY) +
     (200-OTHOURS * 200-HRLY-RATE).

2200-PRINT-REPORT.
  IF 100-88-UNION-EMPLOYEE
     PERFORM 2210-PRINT-UNION
  ELSE
     PERFORM 2220-PRINT-NONUNION

2210-PRINT-UNION.
  MOVE 100-LAST-NAME          TO  400-LAST-NAME.
  MOVE 100-FIRST-NAME         TO  400-FIRST-NAME.
  MOVE 100-SSN                TO  400-SSN
  MOVE 100-JOB-CODE           TO  400-JOB-CODE.
  MOVE 100-SALARY             TO  400-SALARY.
  MOVE 200-HOLD-HOURS         TO  400-HOURS.
  MOVE 200-TOTPAY             TO  400-TOTPAY.
  MOVE 200-OTPAY              TO  400-OTPAY.
  WRITE REPORT-RECORD FROM 400-PRINT-DETAIL-LINE
                AFTER ADVANCING 2 LINES.
```

FIGURE 4.0 continued

```
2220-PRINT-NONUNION.
  MOVE 100-LAST-NAME          TO  400-LAST-NAME.
  MOVE 100-FIRST-NAME         TO  400-FIRST-NAME.
  MOVE 100-SSN                TO  400-SSN
  MOVE 100-JOB-CODE           TO  400-JOB-CODE.
  MOVE 100-SALARY             TO  400-SALARY.
  MOVE 200-HOLD-HOURS         TO  400-HOURS.
  MOVE 200-TOTPAY             TO  400-TOTPAY.
  MOVE ZEROES                 TO  400-OTPAY.
  WRITE REPORT-RECORD FROM 400-PRINT-DETAIL-LINE
              AFTER ADVANCING 2 LINES.

3000-TERMINATION
*  CLOSE FILES AND STOP RUN.

8000-READ-EMPLOYEE-REC.
*  READ RECORD FROM FILE
*     AT THE END OF THE FILE SET 200-END-OF-FILE TO "YES"
```

FEWER FUNCTION NAMES TO REMEMBER AND COORDINATE

Nearly all programming shops have several different types of functions. Most of these functions are very different from one another. But in many other cases, the functions are very similar with only slight variations of the data types or operations. There are also occasions where a function will act almost like another function but coded with a different programming style. Sometimes these functions will be in their own separate library file and other times the functions will be within the source code for the entire program. The result of too many functions is an overcrowded and difficult to maintain library. It is also difficult for a programmer to take advantage of using any of the common functions because of a large number to choose from and because the names may not always clearly relate to what the function is actually doing.

When designing the structure of the COBOL program most of today's programmers would apply a structured programming technique called cohesion. **Cohesion** means that all paragraphs or sub-programs should perform only one task. With cohesion, the paragraph or sub-program will be less complicated and easier to follow since the programmer must concentrate on only one task. This can help during the maintenance process. Naming these paragraphs or sub-programs is also very important so that the maintenance programmer can easily identify what is being done. Since there are two types of employees in our example, there are two separate ways to calculate their weekly pay amount and two separate print lines. We would want to code each of these similar tasks as separate paragraphs. Since these paragraphs will perform very similar functions except that one is done for union and one is done for non-union, their paragraph names were made very similar.

Each of the paragraphs in FIGURE 4.0 performs only one task. Even though the calculation of a week's pay seems to be one task, the task is done differently depending on whether the employee is union or non-union. In this particular example there are only two print paragraphs and two calculate pay paragraphs so this is not really too confusing. Imagine if there were 10 different types of employees with separate calculations and print detail formats. This would eventually leave the programmer with too many similar paragraphs to keep track of and maintain. Other programs that may need to perform payroll calculations would also have to repeat the same paragraphs.

If a function such as calculate payroll will be used by many different programs, the paragraphs may be removed from the code and treated as callable sub-programs. Even if the functions were treated as sub-programs, there would probably still be a separate sub-program for each type of employee. In our example below, 100-SALARY and 100-HOURS pass the required information into the sub-program and the results are passed in the parameters 200-OTPAY and 200-TOTPAY. The parameters to pass information into and out of the sub-program follow the USING statement. COBOL will only allow 8 position unique names for the sub-programs, so to calculate pay for union employees the name CALCUNON was chosen. To calculate pay for non-union employees the name CALCNONU was chosen. The calculate pay paragraph that follows would replace the same paragraph name in FIGURE 4.0. Paragraphs 2110 would be moved to the sub-program CALCUNON and the paragraph 2120 would be moved to the sub-program CALCNONU.

```
2100-CALCPAY.
  IF 100-88-UNION-EMPLOYEE
     CALL "CALCUNON" USING    100-SALARY
                              100-HOURS
                              200-OTPAY
                              200-TOTPAY

  ELSE
     CALL "CALCNONU" USING    100-SALARY
                              100-HOURS
                              200-OTPAY
                              200-TOTPAY.
```

By looking at the names of the sub-programs, it is not obvious what will happen. The name CALCUNON and CALCNONU will not mean much

to a maintenance programmer. The development programmer must remember to put good paragraph names or add good comments to explain just what is going on in the sub-programs. Like the problem with paragraphs, when there are 10 different types of employees there will need to be 10 different sub-programs. This can be too much for the maintenance programmer to try to keep straight.

Another technique exists to use a sub-program but code it as one large sub-program called CALCPAY. An additional parameter would have to be passed to the program identifying what type of employee is being processed. Paragraph 2100 as it appears in FIGURE 4.0 would be replaced with the following paragraph.

```
2100-CALCPAY.
  CALL "CALCPAY" USING        100-SALARY
                              100-HOURS
                              100-FAMILY
                              200-UNION-OTRATE
                              200-OTPAY
                              200-TOTPAY.
```

Paragraph 2100, 2110 and 2120 as they appear in FIGURE 4.0 would move to the sub-program CALCPAY. By using one single sub-program, the main program is a little less confusing since it does not have a separate paragraph for each type of calculation. Only one function name must be remembered to calculate the pay. The only drawback is that now the sub-program of CALCPAY will contain several paragraphs that perform similar calcuations but have slightly different names.

The use of paragraphs or callable sub-programs are language features, but using them cohesively is a programming technique. The programmer must remember to have only one task for each paragraph, choose a meaningful name and add sufficient comments so the maintenance programmers job is made easier. With POP you have a lot of function names to remember and coordinate.

ACCIDENTAL MODIFICATIONS

A programmer must be very careful when making modifications to a program. In POP it is very easy to accidentally modify data variables that should not have been modified.

In COBOL, every field in working storage is accessible to every line in the program. This would mean that a programmer may move data to the wrong field and no warning would be given at compile time. These errors are usually found during testing or once the program is running in production.

Lets assume the following move statement was accidentally put in paragraph 2120 of FIGURE 4.0.

```
MOVE 200-OTPAY  TO  200-UNION-OTRATE
```

The program would compile with no errors since both fields 200-OTPAY and 200-UNION-OTRATE exist in working storage. The

problem is that 200-UNION-OTRATE should not be changed. In COBOL there is no way to prevent a number from being moved to 200-UNION-OTRATE.

Some POP languages like COBOL have no language features or techniques that can be used to prevent accidental modifications.

ERROR DETECTION

If a program is not working properly, the program is said to have a "bug". It is then the job of the maintenance programmer to correct the bug. This task can sometimes be very difficult especially when many different functions have access to the same variable data fields. In our COBOL example of FIGURE 4.0, lets assume that a maintenance programmer was told, the union employees overtime pay is not being calculated correctly. The input file containing the employee's hours, salary and job code was found to be accurate so the problem must exist in the program. The programmer would probably first look at paragraph 2110 and see if the formula is correct. When the formula is reviewed, it is found to contain several working storage fields. Each of these fields would have to be evaluated to determine if they were being improperly set. Since every field in working storage is accessible to every line in the program, there may be a lot of work ahead of the programmer to determine what might have gone wrong with the fields that make up the formula. If for example, the program in FIGURE 4.0 was accidently

modified so that the following move statement was put in paragraph 2120, it might take a while to find this bug.

MOVE 200-OTPAY TO 200-UNION-OTRATE.

The working storage field for 200-UNION-OTRATE is supposed to be a constant value of 1.5 which is correct. But the error of adding this move causes the constant 200-UNION-OTRATE to change every time paragraph 2120 is executed. Since the union employees pay is incorrect, the maintenance programmer would probably not even look at the non-union paragraph for quite a while.

Most POP languages such as COBOL rely on tools such as debuggers to detect this type of problem. In programming shops where debuggers do not exist, the programmer must step through the code line by line to look for logic flaws. Stepping through a program line by line can be very time consuming and therefore costly during the maintenance process.

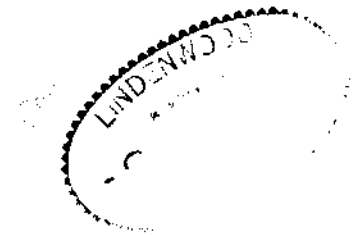
CHANGE CONTROL

When a maintenance programmer is making a change to part of a program, it can be difficult to determine what impact that change will have on the entire program. As identified early in this chapter, the program in FIGURE 4.0 was developed using the structured programming technique of cohesion. The example bug found in the

Error Detection section above can be corrected fairly easily once it is found. Since this bug is in a cohesively designed program, we can safely say that setting the union overtime rate field in the non-union calculate pay paragraph does not make sense. In fact having this bug in the program actually made paragraph 2120 be non-cohesive since the paragraph did more than just calculate non-union employee's weekly pay. By removing the statement in error, paragraph 2120 returns to a cohesive state and in this case, nothing else would be impacted by the change. When a program was written using cohesion, and the programmer must later make a change, then it is fairly easy to determine where the change should be made.

Some changes that are requested can have a major impact regardless of whether programs were developed with cohesive techniques. The President of CASE Associates Incorporated presented a paper at Showcase VI stating that one of the disadvantages of Structured Methods is that even small programming changes can lead to high maintenance costs.[29] Assume that management plans to restructure the employee's job code. It will no longer be possible to look at positions 3 through 5 of the job code to determine if the employee is a union employee. Management has proposed that the positions 1 through 2 of job code still show the employee's grade level but now positions 3 through 4 will be used to tell

[29]William David Sharon, President of CASE Associates, Inc., "Comparing Object-Oriented and Structured Methods", Showcase VI, September 25, 1991, St. Louis, MO.



in which field the employee is working and position 5 will be used to match against a table to determine the employee family. For example a person with a new job code of "52PG1" would be identified as follows :

positions 1-2 ; value of 52 ; means grade level of 52
positions 3-4 ; value of PG ; means programming field
position 5 ; value of 1 is matched to a table to find that the employee is in union 100

The impact of this change on the program in FIGURE 4.0 is moderate. Logic must be added to use position 5 of the job code and read a table to find out if the employee is union.

JOB CODE FIELD

```
05 100-JOB-CODE.
   10 100-GRADE     PIC X(02).
   10 100-FIELD     PIC X(02).
   10 100-FAMILY    PIC X(01).
```

FAMILY TABLE

```
01 FAMILY-TABLE.
   10 FAMILY-KEY    PIC X(1).
   10 FAMILY-CODE   PIC X(3).
```

The IF statements in paragraphs 2100 and 2200 will have to be modified to test the result of the table search.

```
2100-CALCPAY.
   SEARCH FAMILY-TABLE
      WHEN FAMILY-KEY = 100-FAMILY
         THEN MOVE FAMILY-CODE TO 200-HOLD-FAMILY.

   IF 200-HOLD-FAMILY = "100" OR 200-HOLD-FAMILY = "200"
      PERFORM 2110-CALCPAY-UNION
   ELSE
      PERFORM 2120-CALCPAY-NONUNION.
```

But even more significant is that all the programs in the company that use the family identifier within the jobcode will have to be modified with the same changes that applied to the sample program in FIGURE 4.0. This could take a significant amount of time. POP languages such as COBOL

do not have programming language features or techniques that eliminate this type of maintenance problem.

SUMMARY

In summary the procedure oriented languages do not have inherent abilities to (1) reduce confusion of having many similar functions, (2) prevent accidental modifications, (3) help detect errors quickly or (4) limit the impact of changes to a system. There is no way to prevent the accidental modification of data in POP languages such as COBOL. If the development programmer used a structured programming technique of cohesion, error detection and code changes will be a little easier for the maintenance programmer. The only way to reduce the confusion of having many similar functions is to have good documentation to easily find and understand the purpose of each function. Relying on the programmer to understand and remember to apply these techniques is not a perfect solution to the problem but techniques do help in program maintenance if consistently applied. But the reality is that programmers are often under a lot of time pressure and often leave out the documentation or document poorly[30] and don't always follow the rules of cohesion and good programming techniques.

[30]Programmers from McDonnell Douglas, AT&T, and Home Savings of America said that they felt pressured to meet programming deadlines and often neglected to update or write good documentation.

CHAPTER 5

MAINTENANCE PROGRAMMING USING OOP

One of the reasons for the development of object oriented languages was to improve program maintenance.[31] The result was a language style with features specifically designed to help the four program maintenance areas listed in FIGURE 3.0.

In chapter 4, Company A requested that a program be written to calculate the weeks pay and print a formatted report of the pay detail information. Chapter 4 showed that program in the POP language of COBOL. That same program will now be shown using the OOP language of C++. To reduce the length of the program, only the important lines of code will be included. Comments can be found between the symbols /* and */ or following the symbol // when the comment is on the same line as source code.

---

[31]Dick Pountain, "Object-Oriented Programming", BYTE, February 1990, page 257.

FIGURE 5.0 - Employee Payroll Report in C++

```
#include <iostream.h>
#include <string.h>
 /* Beginning of Class Employee definition */
 enum boolean {false, true};          // false = 0 ; true = 1

 class Employee {  // Class Employee definition
 private:          // Accessible only to this class
  char *last_name;  // pointer to last name
  char *first_name; // pointer to first name
  char *job_code;   // pointer to job code
  char grade[2];    // 2 character string for the job grade
  char family[3];   // 3 character string for job family
  long soc_sec_no; // Long int. value for social security number
  boolean in_union;// Boolean that tells if employee is in a union
public:           // methods accessible by all classes
/* If 3 string and 1 numeric parameter is passed to the Employee */
/* constructor, the private fields will be set equal to values passed */
  Employee (char LastName[20], char FirstName[15],
            char JobCode[5], long ssn)
  {
   last_name = LastName; // Copies name parm to private data item
   first_name = FirstName;  // Copies name parm to private item
   job_code = JobCode;   // Copies job code parm to private item
   soc_sec_no = ssn;     // Copies soc sec no parm to private data item
   /* Job grade is the same as the first 2 pos of job code */
   grade[0] = job_code [0]; // Position 1 of job code to pos 1 of grade
   grade[1] = job_code [1]; // Position 2 of job code to pos 2 of grade
   /* Job family is the same as the last 3 pos of job code */
   family[0] = job_code[2]; // Pos 3 of job code to pos 1 of family
   family[1] = job_code[3]; // Pos 4 of job code to pos 2 of family
   family[2] = job_code[4]; // Pos 5 of job code to pos 3 of family
  }
```

FIGURE 5.0 continued

```
/* Returns true if the employee is union otherwise returns false */
boolean IsUnion ()
{
  if (strncmp (family,"100",3)==0)
     in_union = true;
  else
  { if (strncmp (family,"200",3)==0)
          in_union = true;
     else
          in_union = false;
     };
  return in_union;
}

/* Prints 3 strings and 1 long integer */
/* which are last name, first name, job code and ssn */
void Print ()
{ cout << last_name << "  " << first_name << "  " << soc_sec_no
       << "  " << job_code  << "  ";
}
};                    // End of Class Employee Definition

class UnionEmp : Employee {     // Class UnionEmp definition
private:                // Accessible only to this class
  double salary;        // holds salary for a union Employee
  double hours;         // holds hours worked for a union employee
  double tot_hours;     // holds total hours
  double otpay;         // holds the calculated overtime pay amount
  double totpay;        // holds the calculated total pay amount
  double othours;       // holds the calculated overtime hours
  double otrate;        // holds overtime rate
public:                 // methods accessible by all classes
/* If 3 string and 3 numeric parameter is passed to the UnionEmp */
/* constructor, the private fields will be set equal to the values passed */
UnionEmp(char LastName[20], char FirstName[15],
          char JobCode[5], long ssn, double sal, double hrs) :
   Employee (LastName, FirstName, JobCode, ssn)
{   otrate = 1.5;
    salary = sal;
    hours = hrs;
    tot_hours = hours;
}
```

FIGURE 5.0 continued

```
void CalcPay()
{ othours = hours - 40;
 if (othours < 0 )
    othours = 0;
 else
    hours = 40;
 otpay = (othours * otrate * salary);
 totpay = (hours * salary) + otpay;
}

void Print()
{
 Employee :: Print();
 cout << salary << " " << tot_hours << " " << totpay << " "
      << otpay << "\n";
}
};        // End of class UnionEmp definition

class NonUnionEmp : Employee {  // NonUnionEmp definition
 private:              // Accessible only to this class
  double salary;       // holds salary for a non-union employee
  double hours;        // holds hours worked for a non-union employee
  double tot_hours;// holds total hours
  double otpay;        // holds the calculated overtime pay amount
  double totpay;       // holds the calculated total pay amount
  double othours;      // holds the calculated overtime hours
  double hrly_salary;  // holds hourly salary
 public:               // methods accessible by all classes
 /*  If 3 string and 3 numeric parameter is passed to this constructor, */
 /*  the private fields will be set equal to the values passed */
 NonUnionEmp  (char LastName[20], char FirstName[15],
               char JobCode[5],  long ssn, double sal, double hrs) :
          Employee (LastName, FirstName, JobCode, ssn)
          {      salary = sal;
                 hours = hrs;
                  hrly_salary = hours / 40;
          }
```

FIGURE 5.0 continued

```
void CalcPay ()
 {othours = hours - 40;
  if (othours < 0 )
     othours = 0;
  else
     hours = 40;
  otpay = (othours *  hrly_salary);
  totpay = (hours * hrly_salary) + otpay;
 }

void Print ()
 {
   Employee::Print ();
  cout  << salary << " " << tot_hours << " " << totpay << "\n";
 }

 };                    // End of class NonUnionEmp definition

/*  Start of main program logic */
main ()
 {
  char new_ln[20], new_fn[15], new_jcd[5];
  double new_sl, new_hr;
  long new_ssn;
  boolean uniontest;
```

FIGURE 5.0 continued

```
do {     // start of loop through employee database

/*  Construct a NewEmp object from the Employee class.      */
Employee  NewEmp(new_ln, new_fn, new_jcd, new_ssn);

/*  only print information on union employees   */
uniontest = NewEmp. IsUnion();
if (uniontest == true)
 {
   UnionEmp  NewUnionEmp (new_ln, new_fn, new_jcd,
                                        new_ssn, new_sl, new_hr);
   NewUnionEmp.CalcPay ();
   NewUnionEmp. Print();
 }
else
{
  NonUnionEmp NewNonUnionEmp (new_ln, new_fn,
                          new_jcd, new_ssn, new_sl, new_hr);
   NewNonUnionEmp. CalcPay ();
   NewNonUnionEmp. Print ();
 }
 }
while . . . ;  // employee records remain

}
```

In this example program, the root class is named **Employee** and derived from the root class **Employee** are the two sub-classes of **UnionEmp** and **NonUnionEmp**. All of the methods (eg, ***Employee***, ***CalcPay***, and ***Print***) in the public section of the class **Employee** are inherited by both class **UnionEmp** and class **NonUnionEmp**.

What follows are the four program maintenance areas listed in Figure 3.0.

FEWER FUNCTION NAMES TO REMEMBER AND COORDINATE

Most OOP languages allow methods within different classes to have the same function name. In the example program in FIGURE 5.0, all three classes have a method ***Print***. Both the class **UnionEmp** and class **NonUnionEmp** have a method ***Calcpay***. In a program, when a message is sent, the system is able to determine which method actually needs to be executed based on the object associated with the message. For example, in the main program there are two lines of code that send the message ***Calcpay*** as follows:

NewUnionEmp. ***Calcpay***();
and
NewNonUnionEmp. ***Calcpay***();

Object **NewUnionEmp** is an instance of class **UnionEmp**. Object **NewNonUnionEmp** is an instance of class **NonUnionEmp**. The same message of **Calcpay** is sent to the different classes and the system takes care of executing the correct method. To calculate pay for all possible types of employees, the programmer has to remember only one message. This capability is known as **polymorphism** and helps to simplify program maintenance. Polymorphism is a feature of the language and is more natural for the programmer than trying to make up new names for each possible variation on the same basic function.

ACCIDENTAL MODIFICATIONS.

Care should be exercised when making modifications to a program so that problems do not result from the modifications. OOP languages have a feature of **encapsulation** that helps prevent accidental modifications. Encapsulation limits access to the data and methods in a class. By limiting the access, fewer classes and programs will have the ability to modify data. Without access to data, it is impossible to accidentally modify the data.

Lets take for example, the problem similar to the one described in the error detection section of chapter 4 on maintenance in POP. The programmer has been notified that the payroll calculation detail report is showing incorrect overtime amounts for all union employees. In our example in chapter 4, the error was a move in a non-union calculate payroll paragraph that accidently reset the overtime rate field (**otrate**) for union employees. In our example in FIGURE 5.0, this would not be possible. The data field **otrate** is in the private area of the class **UnionEmp** which means that no class outside of the class **UnionEmp** has the ability to change this field. This language feature helps prevent accidental modifications in OOP languages.

The programmer can cause errors in object oriented programs, but the chances are significantly lower because of encapsulation. If a program or class cannot access certain data, it would not be able to accidently modify that data.

ERROR DETECTION

Finding errors in programs can be a difficult and time consuming task. Fortunately the OOP language feature of **encapsulation** has helped decrease the time it takes to find errors. Encapsulation, both groups the data and methods together into one class and provides controlled access to the data and methods which helps in error detection.

For an example of a program bug for the object oriented program in FIGURE 5.0, suppose there is an error in the method ***CalcPay*** in the class **UnionEmp**. The formula contains a typographical error. The statement should have read as follows :

totpay = (hours * salary) + otpay

Instead, the leading "t" was left off the field totpay as follows :

otpay = (hours * salary) + otpay

The method that prints the union detail report line is found in the class **UnionEmp**. The fields that are printed on the report are either passed to the constructor method ***UnionEmp*** or they are calculated values within the class **UnionEmp**. Since the field printed in error was both totpay and otpay, the only two possibilities are that either incorrect information was passed to the constructor or that methods and/or data within the class **UnionEmp** definition are incorrect. The maintenance programmer would probably first look at the class **UnionEmp** and the method ***CalcPay*** and should then see the mistake.

With encapsulation, the programmer is more quickly able to narrow down the possible locations that may have caused a problem. This certainly reduces the time spent on error detection and correction which reduces program maintenance time. Encapsulation is a language feature that is a natural part of programming in OOP languages.

CHANGE CONTROL

Encapsulation is also a helpful language feature in relation to program changes.[32] With encapsulation, since data and method access is controlled by the class, it is fairly easy to determine what impact there would be if a change must be made to the class.

As an example, lets assume the change as described in the change control section of chapter 4 where the job code field positions 3 through 5 can no longer be used directly to determine whether or not an employee is union. In FIGURE 5.0 the class **Employee** has a method ***IsUnion*** that must be modified to accept the new method of determining if an employee is union. Assuming that all object oriented programs within the company use the class **Employee** and the method ***IsUnion***, no programs other than the class **Employee** should have to be modified. The code modifications are isolated to one method in one class. No

[32]One programmer from McDonnell Douglas specifically mentioned encapsulation as being very effective at localizing programming changes.

programs outside the class **Employee** had to understand the way the class **Employee** determines an employee was union. This change which can be a major impact in POP languages was a minimal change to this object oriented program. The new class definition would look as found in FIGURE 5.1 with the changes in bold and italics.

FIGURE 5.1 - Employee Payroll Report in C++ with Job Family change

```
#include <iostream.h>          // I/O areas
/*  Beginning of Class Employee definition  */
enum boolean {false, true};   // false = 0 ; true = 1
class Employee {              // Class Employee definition
private:                      // Accessible only to this class
 char *last_name;             // pointer to  last name
 char *first_name;            // pointer to  first name
 char *job_code;              // pointer to job code
 char grade[2];               // 2 character string for job grade
 char field[2];               // 2 character string for job field
 char fam_indx                // 1 character string for family table index
 char family[3];              // 3 character string for family from table
 long soc_sec_no;             // Long int. value for social security number
 boolean in_union;            // Boolean that tells if employee is in union
public:                       // methods accessible by all classes
 /*  If 3 string and 1 numeric parameter is passed to this constructor, */
 /*  the private fields will be set equal to the values passed */
 Employee(char ln[20], char fn[15], char jcd[5], long ssn)
 {
   last_name = ln;            // Copies name parm to private data item
   first_name = fn;           // Copies name parm to private data item
   job_code = jcd;            // Copies job code parm to private data item
   soc_sec_no = ssn;          // Copies soc sec no parm to private data item
   /*  Job grade is the same as the first 2 positions of job code  */
   grade[0] = job_code [0];  // Position 1 of job code to position 1 of grade
   grade[1] = job_code [1];  // Position 2 of job code to position 2 of grade
```

FIGURE 5.1 continued

```
    /*  Job family is the same as the last 3 positions of job code  */
    field[0] = job_code[2];  // Position 3 of job code to pos. 1 of family
    field[1] = job_code[3];  // Position 4 of job code to pos. 2 of family
    fam_indx[0] = job_code[4]; // Position 4 of job code to pos. 2 of family
}
/*  Returns true if the employee is union otherwise returns false  */
boolean IsUnion();
  {
  /* code goes here to search the family table for a match  */
  /*  on the family code.  The value from the table search  */
  /*  is moved to the field family  */
     if (strncmp (family,"100",3) == 0 )
         in_union = true;
     else
         {
         if (strncmp (family,"200",3) == 0 )
             in_union = true;
         else
             in_union = false;
         };
     return in_union;
  }

/*  Prints  3 strings and 1 long integer   */
/*  which are last name, first name, job code and social security number*/
void Print ()
{
    cout <<  last_name << " " <<  first_name << " "
         << soc_sec_no << " " << job_code  << "\n" ;
}

};                  //  End of Class Employee Definition
```

SUMMARY

In summary, object oriented languages have language features that help reduce program maintenance time. **Polymorphism** helps reduce the work that a programmer must do to determine which of the similar functions is needed to perform a task. Several similar functions acting on different classes can all have the same name (i.e., **function overloading**) and the system will determine for the programmer which of the functions should be executed. **Encapsulation** helps prevent accidental modification of data since classes and programs are restricted from accessing many data items in other classes. With no access, it is impossible to modify the data. **Encapsulation** helps the programmer detect errors quickly since the error is isolated and the programmer can quickly narrow the possible areas that would have had access to the data or method. **Encapsulation** is also helpful when making changes to a class. Usually only the class and its sub-classes will require a change. Other classes and programs usually require no change at all. Since polymorphism, encapsulation and function overloading are features of the OOP language, they will be used by the programmer more naturally then forcing the use of a programming technique.

CHAPTER 6

CODE REUSE OVERVIEW

Code reuse can be accomplished in many different ways. Anytime code from an existing application is used to develop a different application, code reuse has taken place. Some examples are:

(1) Use existing functions by copying the code into the program. The common function becomes part of the new source program.

(2) Use the copy statement to copy the source in at compile time. The common function is not part of the new source program but is part of the object code for the new program.

(3) Use the call statement to run common functions from a run-time library. Both source and object for the common function are separate from the new program calling the common function.

(4) Develop new code that inherits data and methods from existing code.[33]

The above examples can be accomplished with programming techniques and language features available to both procedure oriented programming (POP) and object oriented programming (OOP).

During my research for this thesis, I talked with several programmers who said there was an unspoken understanding in their organization that code reuse was expected of them. In some cases the programmers even received training in the area of code reuse.[34]

---

[33] Daniel G. Bobrow, "The Object of Desire", Datamation, 1 May 1989, page 38.

[34] At least one programmer from both McDonnell Douglas and AT&T had attended a company sponsored class where code reuse was part of the training.

Only the programmers using OOP felt that code reuse has been successfully used, in their work environment.[35] Regardless of the programming style, there are some general reasons to explain why code reuse is either not working or why it is only moderately successful. Some of those explanations are:[36]

(1) The programmer is not aware that a common function exists or is not able to easily find the name of the common function.

(2) The programmer has a lack of confidence in the existing common function.

(3) The function needs to be slightly modified to suit the new code and rather than modifying the existing function to make it more generic, a new function is created.

As we have seen, program maintenance is the major cost for programming shops today. Code reuse will not only help reduce development time on new software but it can also help reduce maintenance costs on old software for the following reasons:[37]

(1) Shared common functions result in fewer lines of code to maintain.

(2) The accuracy of the reused code is greater than the accuracy of programs coded from scratch. Reused code will have gone through more complete testing then the code developed from scratch. The more a piece of code is reused, the better the chances for working out all of the bugs.

---

[35] At least one programmer from McDonnell Douglas, AT&T, and Computer Artisans felt that code reuse has been successful from them.
[36] Ted J. Biggerstaff and Alan J. Perlis, Software Reusability : Applications and Experience, Addison-Wesley Pub. Co., 1989, page 2.
[37] Will Tracz, Software Reuse : Emerging Technology, IEEE Computer Society Press, 1988, page 35.

(3) Programs will follow a standard interface to a piece of reusable code. A new piece of reusable code will have the same interface to the program as the old code. Here we only have to recompile the module that we changed and not each of the programs that use it.

Code reuse is an important part of the fight to reduce the cost of program maintenance. The options available to promote code reuse are the same options available to reduce program maintenance.

(1) Use *programming tools* to improve location and retrieval of common functions.

(2) Make sure programmers use the *language techniques and features* available for the chosen language.

(3) *Choosing a different language or style of language* to find one better able to promote code reuse.

PROGRAMMING TOOLS

One of the problems mentioned earlier is that programmers are not aware of existing common functions or they do not know the names of the functions. This can be partially solved by the use of commercially available library tools. Many of these tools have search capabilities to help in locating the function needed by a programmer. Without this type of tool it may be difficult to store and easily find a common function that will fit the needs of a new application. The programming tools that allow storage and easy retrieval of common functions will help only if these functions are kept in libraries accessible to other programmers.

A person should always program as though the code being developed will probably be reused. If a function being developed has potential for being reused, the development programmer should design the function so it can be easily reused. The design should include the following:

(1) Plan for future problem domains
(2) Good documentation

Plan for future problem domains

The programmer must use planning to develop code that can be easily reused. The programmer must think ahead about what pieces of information in the current problem domain may not be exactly what will be needed by any future problem domains. If some of this variable information is passed into the program using parameters rather than hardcoding the information, the code will be usable for more applications than it would have been with the hardcoding.

Good documentation

Good documentation is important in code reuse. Regardless of the language style and the method of reuse, the development programmer needs to document each of the following that applies :

(1) The purpose of the task (implemented as a function or procedure.)
(2) The variables that must be defined.
(3) The type and number of parameters that must be used.
(4) The data and methods if any that can be inherited.

This constitutes a clear definition of the formal and actual parameters that allow communication between a main program or a sub-program and a function or procedure. This type of documentation called an interface specification, is a technique that can be applied to the language. It can help other programmers decide if a function or procedure will fit their programming need and at the same time provide information to the programmer on how to use the function or procedure.

There are some inherent language styles that promote code reuse by providing language features[38] such as inheritance and templates in OOP. It is much better to have language features that promote the reuse of code rather than forcing the programmer to remember to apply techniques to the language. Even though a language feature may be more successful for code reuse than a programming technique, they can be used together to more strongly support code reuse.

CHOOSING A DIFFERENT LANGUAGE

Since code reuse can help dramatically reduce program development and program maintenance time, it is strongly suggested that a programming shop use the language best able to promote code reuse. If code reuse has not improved after using the tools, techniques and features of the chosen language, the programming shop should consider choosing a different language. This may be as minor as going from one

[38] Ted J. Biggerstaff and Alan J. Perlis, Software Reusability : Concepts and Models, Addison-Wesley Publishing Company, 1989, page 36.

procedure oriented language to another or as major as going from a procedure oriented language to an object oriented language.

SUMMARY

There will always be some programmers that will resist code reuse, but for the most part, if programmers are using a language that has features to inherently promote reuse, it will start to happen since less effort is required of the programmer. In summary, chapters 7 and 8 will deal with the differences between OOP and POP that causes OOP to be more successful at code reuse. FIGURE 6.0 lists two areas where code reuse is handled differently in OOP and POP.

FIGURE 6.0

(1) Writing reusable code.
(2) Making minor changes to reusable code.

CHAPTER 7

CODE REUSE IN POP

Code reuse in procedure oriented languages has not been very successful.[39] This style of language does not inherently promote code reuse. Instead, the demand is on the programmer to use techniques to reuse existing code. This chapter will look at procedure oriented programming (POP) and how it addresses the two areas of code reuse as identified in FIGURE 6.0.

To continue our payroll example from Figure 4.0 in Chapter 4, a common function could be written to calculate the weekly pay. This is especially useful if other programs will need to do the same calculation and can take advantage of reusing this function.

WRITING REUSABLE CODE.

To write reusable code, the programmer must try to keep the function general enough so that it can handle simple variations of the main function. For example, a calculate pay function must be able to

---

[39]This seems to be agreed upon by current articles and programmers actually trying to reuse POP code.
Grant Buckler, "OOP is more than a buzzword", Computing Canada, September 1991, page 31.
All programmers using POP agreed that code reuse is either moderately effective or not very effective in their organization. These programmers were from JWP Controls, McDonnell Douglas, Home Savings of America, and Tripos.

handle different salaries, salary formats, hours worked and overtime rates by accepting these values as variables or parameters. In our problem domain, the salary for union employees is stored in an hourly format and the non-union employee's salary is stored in a weekly format. To be able to determine the salary format, the job family will also be treated as an input variable or parameter. A technique to naming variables in COBOL is to prefix the name with 100 or 200 to distinguish an input variable or parameter from an output variable or parameter. The input variables or parameters will be as follows:

(1) The salary will be named 100-SALARY.
(2) The hours worked will be named 100-HOURS.
(3) The job family will be named 100-FAMILY.
(4) The overtime rate will be named 200-OTRATE.

The output of the function would be the calculated pay for the employee. Output variables or parameters will have the prefix of 300.

(5) The total pay will be named 300-TOTPAY.
(6) The overtime pay will be named 300-OTPAY.

COBOL does not distinguish between input and output parameters. There is no way in COBOL to prevent the function from making changes to the input variables.

FIGURE 7.1 below shows the calculate pay paragraph as reusable COBOL code.

FIGURE 7.1 - Calculate Pay COBOL paragraph in reusable code

```
0000-CALCPAY.

 IF 100-88-UNION-EMPLOYEE
      MOVE 100-SALARY TO  200-SALARY
 ELSE
      COMPUTE 200-SALARY = 100-SALARY / 200-FULLTIME-HOURS.

 COMPUTE 200-OTHOURS = 100-HOURS - 200-FULLTIME-HOURS.
 IF 200-OTHOURS < 0
      MOVE 0 TO 200-OTHOURS
      MOVE 100-HOURS TO 200-HOLD-HOURS
 ELSE
      MOVE 200-FULLTIME-HOURS  TO 200-HOLD-HOURS.

 COMPUTE 300-OTPAY =
      200-SALARY * 100-OTRATE * 200-OTHOURS.
 COMPUTE 300-TOTPAY =
      200-SALARY * 200-HOLD-HOURS + 200-OTPAY.
```

In POP, and in particular COBOL, the programmer will choose one of the following formats of making this code available for reuse[40] :

(1) COPY command.
(2) CALL command.

Copy command

With the source code for a COBOL function stored in a source library, the programmer can use the COPY command to bring the source into the program at compile time. This copy member usually does not contain all of the divisons required to make a complete COBOL program. It is meant to be copied into a program that already has the required

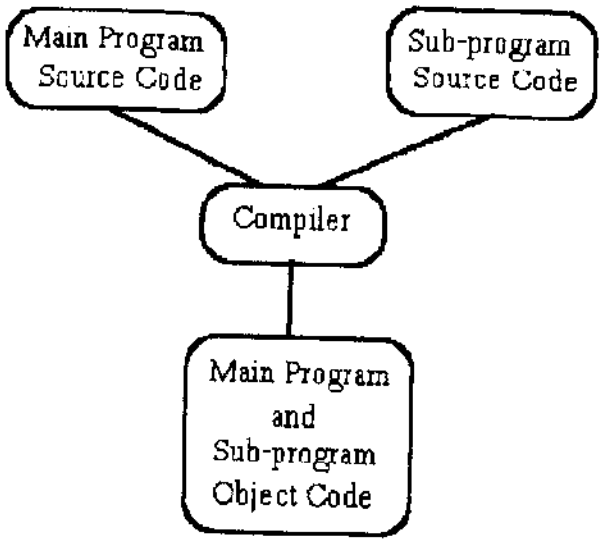
[40]Tom Caldwell, "Putting Effective Maintenance into Practice", Computing Canada, 25 October 1990, page 44.

COBOL divisions. These required divisions include the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION and PROCEDURE DIVISION. The code in FIGURE 7.1 would be stored in a source library with the member name of CALCPAY. All of the variables mentioned in 0000-CALCPAY must be defined in working storage of the main program (i.e. 100-SALARY, 100-HOURS, 100-FAMILY, 200-OTRATE, 300-TOTPAY, 300-OTPAY, ETC.). To copy the source into the main program, the following command should be found in the PROCEDURE DIVISION immediately following the paragraph that performs 0000-CALCPAY. The copy command would look as follows:

COPY CALCPAY FROM *library-name*.

The source for the main program and the source for the CALCPAY function are stored separately but they are brought together at compile time as seen in FIGURE 7.2.

FIGURE 7.2 - **COPY COMMAND COMPILE DIAGRAM**



The copy command is used quite often at McDonnell Douglas Aerospace Information Services with its COBOL programs. In fact, they even wrote a COBOL pre-processor to improve the use of the copy command. The pre-processor allows the programmer to use a file layout that contains asterisks where the numeric prefix would normally be. The preprocessor changes the asterisks to a numeric prefix requested in the copy statement.

COPY EMPLOYEE **PREFIX=100** LIB=PROD

Therefore a line of code ***-NAME, from the file layout EMPLOYEE, would be modified using the preprocessor to be 100-NAME. This allows flexibility to the copy library members.

FIGURE 7.3 below shows a sample main program that uses the CALCPAY reusable code in copy form.

FIGURE 7.3 - Main program that has Copy command

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
FILE-CONTROL
DATA DIVISION.
FILE SECTION.
  FD  REPORT-FILE
        DATA RECORD IS REPORT-RECORD.
  01 REPORT-RECORD                PIC X(133).
  FD  INPUT-FILE
        DATA RECORD IS INPUT-RECORD.
  01 INPUT-RECORD                 PIC X(80).
* The data area is in the working-storage section.
WORKING-STORAGE SECTION.
  01 100-EMPLOYEE-RECORD.
     05  100-LAST-NAME            PIC X(20).
     05  100-FIRST-NAME           PIC X(15).
     05  100-DEPARTMENT           PIC X(04).
     05  100-JOB-CODE.
         10 100-GRADE             PIC X(02).
         10 100-FAMILY            PIC X(03).
            88 100-88-UNION-EMPLOYEE          VALUE "100",
                                                    "200".
     05  100-SSN                  PIC 9(09).
     05  100-SALARY               PIC 9(6)V99.
     05  100-HOURS                PIC 9(3)V99.
```

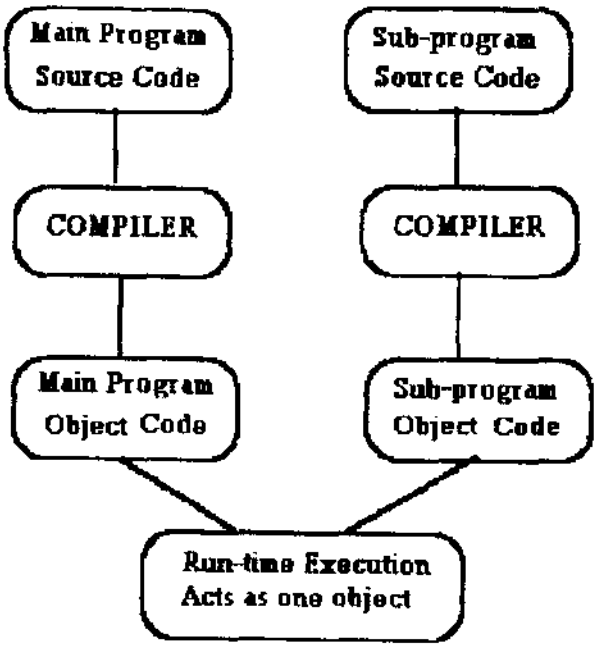
FIGURE 7.3 continued

```
01 200-HOLD-AREAS.
   05  200-OTRATE              PIC 9(3)V99      VALUE 0.
   05  200-HOLD-HOURS          PIC 9(3)V99      VALUE 0.
   05  200-OTHOURS             PIC 9(3)V99      VALUE 0.
   05  200-UNION-OTRATE        PIC 9(3)V99      VALUE 1.5.
   05  200-NONUNION-OTRATE     PIC 9(3)V99      VALUE 1.0.
   05  200-FULLTIME-HOURS      PIC 9(02)        VALUE 40.

01 300-CALCULATED-AREAS.
   05  300-OTPAY               PIC 9(6)V99      VALUE 0.
   05  300-TOTPAY              PIC 9(6)V99      VALUE 0

* The main program and paragraphs are found in the procedure division.
PROCEDURE DIVISION.
0000-MAINLINE.
  PERFORM 1000-INITIALIZATION.
  PERFORM 2000-PROCESS-EMPLOYEE-RECS
     UNTIL 200-END-OF-FILE = "YES".
  PERFORM 3000-TERMINATION.

1000-INITIALIZATION.
*   OPEN FILES, DO INITIAL READ

2000-PROCESS-EMPLOYEE-RECS.
  IF 100-88-UNION-EMPLOYEE
     MOVE 200-UNION-OTRATE TO 200-OTRATE
  ELSE
     MOVE 200-NONUNION-OTRATE TO 200-OTRATE.
  PERFORM 0000-CALCPAY.
  PERFORM 2200-PRINT-REPORT.
  PERFORM 8000-READ-EMPLOYEE-REC.

COPY CALCPAY FROM PRODLIB.

2200-PRINT-REPORT.
*  BUILD AND PRINT REPORT LINE

8000-READ-EMPLOYEE-REC.
*  READ AN EMPLOYEE RECORD

3000-TERMINATION.
*  CLOSE FILES AND DISPLAY COUNTERS
```

Call command

The second format of making code available in a run-time library is the CALL command. A run-time library is a collection of compiled programs. A compiled version of a function can be saved in the run-time library so the object code is available for reuse. The object code for the main program and the object code for the CALCPAY function are stored separately and brought together at run time as seen in FIGURE 7.4.

FIGURE 7.4 - CALL COMMAND COMPILE AND RUN DIAGRAM



Because the compiled main program and the compiled reusable function are stored separately, the data must be passed into the function from the main program through the use of parameters. Careful use of parameters can lead to functions easily reused by a variety of different applications.

COBOL lists parameters in the main program immediately following the USING statement of the CALL command. An example main program that uses a call statement is shown in FIGURE 7.5, shows the parameters passing data into the sub-program are 100-HOURS, 100-SALARY, 100-FAMILY, and 200-OTRATE. The parameter passing data from the sub-program to the main program is 300-TOTPAY and 300-OTPAY.

FIGURE 7.5 - Main program that has Call command

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
FILE-CONTROL.
DATA DIVISION.
FILE SECTION.
  FD  REPORT-FILE
        DATA RECORD IS REPORT-RECORD.
  01 REPORT-RECORD                PIC X(133).
  FD  INPUT-FILE
        DATA RECORD IS INPUT-RECORD.
  01 INPUT-RECORD                 PIC X(80).

* The data area is in the working-storage section.
WORKING-STORAGE SECTION.
  01 100-EMPLOYEE-RECORD.
     05  100-LAST-NAME            PIC X(20).
     05  100-FIRST-NAME           PIC X(15).
     05  100-DEPARTMENT           PIC X(04).
     05  100-JOB-CODE.
         10 100-GRADE             PIC X(02).
         10 100-FAMILY            PIC X(03).
            88  100-88-FAMILY                    VALUE "100",
                                                       "200".
     05  100-SSN                  PIC 9(09).
     05  100-SALARY               PIC 9(6)V99.
     05  100-HOURS                PIC 9(3)V99.
```

FIGURE 7.5 continued

```
01 200-HOLD-AREAS.
   05  200-OTRATE                PIC 9(3)V99      VALUE 0.
   05  200-HOLD-HOURS            PIC 9(3)V99      VALUE 0.
   05  200-OTHOURS               PIC 9(3)V99      VALUE 0.
   05  200-UNION-OTRATE          PIC 9(3)V99      VALUE 1.5.
   05  200-NONUNION-OTRATE       PIC 9(3)V99      VALUE 1.0.
   05  200-FULLTIME-HOURS        PIC 9(02)        VALUE 40.

01 300-CALCULATED-AREAS.
   05  300-OTPAY                 PIC 9(6)V99      VALUE 0.
   05  300-TOTPAY                PIC 9(6)V99      VALUE 0

PROCEDURE DIVISION.
0000-MAINLINE.
  PERFORM 1000-INITIALIZATION.
  PERFORM 2000-PROCESS-EMPLOYEE-RECS
     UNTIL 200-END-OF-FILE = "YES".
  PERFORM 3000-TERMINATION.

1000-INITIALIZATION.
*   OPEN FILES, DO INITIAL READ

2000-PROCESS-EMPLOYEE-RECS.
     IF 100-88-UNION-EMPLOYEE
        MOVE 200-UNION-OTRATE       TO 200-OTRATE
     ELSE
        MOVE 200-NONUNION-OTRATE  TO 200-OTRATE.

     CALL CALCPAY USING  100-SALARY
                         100-HOURS
                         100-FAMILY
                         200-OTRATE
                         300-TOTPAY
                         300-OTPAY.
  PERFORM 2200-PRINT-REPORT.
  PERFORM 8000-READ-EMPLOYEE-REC.

2200-PRINT-REPORT.
*  BUILDS REPORT LINE AND PRINTS.

8000-READ-EMPLOYEE-REC.
*  READ AN EMPLOYEE RECORD

3000-TERMINATION.
* CLOSE FILES AND DISPLAY COUNTERS
```

In the sub-program, parameters are listed following the USING statement in the PROCEDURE DIVISION statement and they are defined in the LINKAGE SECTION . The order of the parameters in the sub-program must match the order of the parameters in the main program. As stated earlier, the compiled version of the CALCPAY function will be stored in a run-time library for reuse. FIGURE 7.6 shows the source code for the sub-program CALCPAY.

FIGURE 7.6 - Sub-program for calculating pay (CALCPAY)

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
FILE-CONTROL
DATA DIVISION.
FILE SECTION.
* The data area is in the working-storage section.
WORKING-STORAGE SECTION.
 01 200-HOLD-AREAS.
    05  200-FULLTIME-HOURS     PIC 9(3)      VALUE 40.
    05  200-HOLD-HOURS         PIC 9(3)V99   VALUE 0.
    05  200-OTHOURS            PIC 9(3)V99   VALUE 0.

LINKAGE SECTION.
  01  100-SALARY               PIC 9(6)V99.
  01  100-HOURS                PIC 9(3)V99.
  01  100-FAMILY               PIC X(03).
        88  100-88-UNION-EMPLOYEE      VALUE "100",
                                             "200".
  01  200-OTRATE               PIC 9(3)V99.
  01  300-TOTPAY               PIC 9(6)V99.
  01  300-OTPAY                PIC 9(6)V99.
```

FIGURE 7.6 continued

```
PROCEDURE DIVISION    USING 100-SALARY
                            100-HOURS
                            100-FAMILY
                            200-OTRATE
                            300-TOTPAY
                            300-OTPAY.
0000-CALCPAY.

 IF 100-88-UNION-EMPLOYEE
     MOVE 100-SALARY TO 200-SALARY
 ELSE
     COMPUTE 200-SALARY = 100-SALARY / 200-FULLTIME-HOURS.

 COMPUTE 200-OTHOURS = 100-HOURS - 200-FULLTIME-HOURS.
 IF 200-OTHOURS < 0
     MOVE 0 TO 200-OTHOURS
     MOVE 100-HOURS TO 200-HOLD-HOURS
 ELSE
     MOVE 200-FULLTIME-HOURS TO 200-HOLD-HOURS.

 COMPUTE 300-OTPAY   = 200-SALARY * 200-OTRATE * 200-OTHOURS
 COMPUTE 300-TOTPAY = 200-SALARY * 200-HOLD-HOURS + 300-OTPAY.
**** GOBACK means to return from the called program
 GOBACK.
```

Comparing copy and call commands.

There are advantages to both the COPY and CALL statements. Each situation must be evaluated separately to determine which method would be best.

Using the COPY command rather than the CALL command will result in faster run-time. With the COPY command, the reusable code is part of the object code for the main program so the system does not have to search for the object form of the reusable code at run-time. With each

CALL statement, the system has to search through a run-time library for the reusable object code. If the function is to be used frequently, the programmer should probably consider making the source available with a COPY statement.

Using the CALL command will make the reusable code more flexible to changes. If a change needs to be made to the reusable code, the code can be modified, recompiled and put back into the run-time library. Most likely no change will need to be made to the programs that reuse this code. If a COPY command was used and the reusable code had to change, each program that uses this reusable code would have to be recompiled. If the reusable code is expected to change very often and if a large number of programs share this code, the programmer would probably want to consider using the CALL statement.

With either method, the function should be cohesive so that it performs only one task and it should be clearly documented so it is clear to all programmers the purpose of the function. Depending on the needs of the application, the reusable code could be made available in either or both methods.

MAKING MINOR CHANGES TO REUSABLE CODE.

Lets assume that a programmer was told that managers would no longer be paid for working overtime hours. Regardless of how many hours worked for the week, a manager will always receive pay equal to one weeks salary. In our example problem domain, we were able to

identify union employees as having a job family of "100" or "200". Now lets assume that managers can be identified as having a job family of "900". All managers will receive their weekly salary regardless of whether or not they work overtime. The reusable code written above in paragraph 0000-CALCPAY will work for the calculation of managers pay with only a few modifications. There are two techniques that will allow the programmer to reuse the above code.

(1) Modify the existing code to work for both the existing and new applications.

(2) Copy the existing code to a new program and modify it to meet the needs of the new application.

Modify the existing code

Modifying existing code has an advantage of fewer lines of code since the parts of the code that are common can be shared by many applications. Less code usually means less maintenance work since there will be fewer untested conditions to cause future problems. If there is only one calculate pay function, a correction would have to be made in only one function rather than multiple functions. Code was added to the

callable function from FIGURE 7.6 and the result can be found in FIGURE 7.7 with the changes in bold letters :

FIGURE 7.7 - Sub-program for calculating pay with Manager modifications

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION
CONFIGURATION SECTION.
FILE-CONTROL
DATA DIVISION.
FILE SECTION.
* The data area is in the working-storage section.
WORKING-STORAGE SECTION.
  01 200-HOLD-AREAS.
     05  200-FULLTIME-HOURS      PIC 9(3)       VALUE 40.
     05  200-HOLD-HOURS          PIC 9(3)V99    VALUE 0.
     05  200-OTHOURS             PIC 9(3)V99    VALUE 0.
LINKAGE SECTION.
  01 100-SALARY                  PIC 9(6)V99.
  01 100-HOURS                   PIC 9(3)V99.
  01 100-FAMILY                  PIC X(03).
     88 100-88-UNION-EMPLOYEE                   VALUE "100",
                                                      "200".
     88 100-88-MANAGER                          VALUE "900".
  01 200-OTRATE                 PIC 9(3)V99.
  01 300-TOTPAY                 PIC 9(6)V99.
  01 300-OTPAY                  PIC 9(6)V99.
```

FIGURE 7.7 continued

```
PROCEDURE DIVISION    USING   100-SALARY
                              100-HOURS
                              100-FAMILY
                              200-OTRATE
                              300-TOTPAY
                              300-OTPAY

0000-CALCPAY.

IF 100-88-MANAGER
    NEXT SENTENCE
ELSE
    IF 100-88-UNION-EMPLOYEE
        MOVE 100-SALARY TO 200-SALARY
    ELSE
        COMPUTE 200-SALARY = 100-SALARY / 200-FULLTIME-HOURS.

IF 100-88-MANAGER
  NEXT SENTENCE
ELSE
  COMPUTE 200-OTHOURS = 100-HOURS - 200-FULLTIME-HOURS
  IF 200-OTHOURS < 0
        MOVE 0 TO 200-OTHOURS
        MOVE 100-HOURS TO 200-HOLD-HOURS
  ELSE
        MOVE 200-FULLTIME-HOURS TO 200-HOLD-HOURS.

IF 100-88-MANAGER
  MOVE 100-SALARY TO 300-TOTPAY
  MOVE ZERO       TO 300-OTPAY
ELSE
  COMPUTE 300-OTPAY =
        200-SALARY * 300-OTRATE * 200-OTHOURS
  COMPUTE 300-TOTPAY =
        200-SALARY * 200-HOLD-HOURS + 300-OTPAY.

GOBACK.
```

Copy existing code to create a new program

The second technique for code reuse is to simply create a new program from the existing program. If a new calculate pay function were created for each type of employee, there would need to be 3 separate

functions that were very similar. We would have a separate function for calculating pay for union, non-union and manager type employees. Each of these smaller and more easily understood functions would be less complicated then the one large function written above. Easy to understand functions are certainly easier to maintain then complicated functions. Another advantage to having separate functions is that a change to one function will not cause bad reactions to the other related but separate functions. Yet a disadvantage is that many smaller but similar functions can be difficult to keep track of and coordinate. One department at McDonnell Douglas has documented standards promoting the concept of copying existing code to create a new program. This concept is only working moderately well because many of the programmers do not take the time to copy and document their code into the code reuse library. Because there is not much code available in the library, many programmers don't even take the time to look there or they often forget it even exists.

SUMMARY

The POP technique of creating separate functions copied from an existing function is the simpliest and probably the most common form of code reuse. It takes a lot of effort for a programmer to develop a generic function that can be reused for multiple applications. Programmers don't want to get involved with modifying a function to make it work for his application and make sure it will still work for all the other applications that are already using it. For this reason programmers will usually just make a copy of the function and modify the copy for the use of the new application. The result is overcrowded libraries that become difficult to maintain. The ideal situation would be to have as few a functions as possible by modifying reusable code when necessary. When the reusable code starts to reach the point of becoming too complicated, a separate function should be written and the existing reusable function should be left as it was. The techniques of modifying existing code or creating a new function can both be used successfully in code reuse. Using either technique is better than coding the function from scratch.

CHAPTER 8

CODE REUSE IN OOP

In object oriented languages, code reuse has been fairly successful.[41] The success of code reuse in this language style is because object oriented programming (OOP) inherently encourages code reuse.[42] This chapter will look at OOP and how it addresses the two areas of code reuse as identified in FIGURE 6.0.

WRITING REUSABLE CODE.

To write reusable code the programmer must plan the design of the code so that many different applications can take advantage of reuse. It is rare that code can coincidentally be reused. In most cases code that is successfully reused was specifically planned for reuse. OOP has features built into the language that help in the design and programming of

---

[41]This seems to be agreed upon by current articles and programmers actually trying to reuse OOP code.
Grant Buckler, "OOP is more than a buzzword", Computing Canada, September 1991, page 31.
Stuart Johnston, "OOP Hailed as Way of the Future", InfoWorld, 15 May 1989, page 17.
All programmers using OOP agreed that code reuse is either moderately effective or very effective in their organization. These programmers were from McDonnell Douglas, AT&T, and Computer Artisans.

[42]Alan Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages", OOPLSA '86 Proceedings, 1986, page 38.

reusable code. In particular the following two OOP language features are useful :

(1) Inheritance
(2) Templates

Inheritance

When a new class is derived from a root class, the new class is said to have inherited data and methods from the root class. Any data and methods found in the protected and public areas of the root class can be reused by the derived class. When data and methods are inherited from the root class, this means that the code defining the data and methods does not have to be repeated in the derived class for the derived class to use them. Inheritance has the following benefits :

(1) Less coding is required since some of the data and methods for the class can be taken from the root class.

(2) Less testing is required since the inherited data and methods have already been tested by other programs.

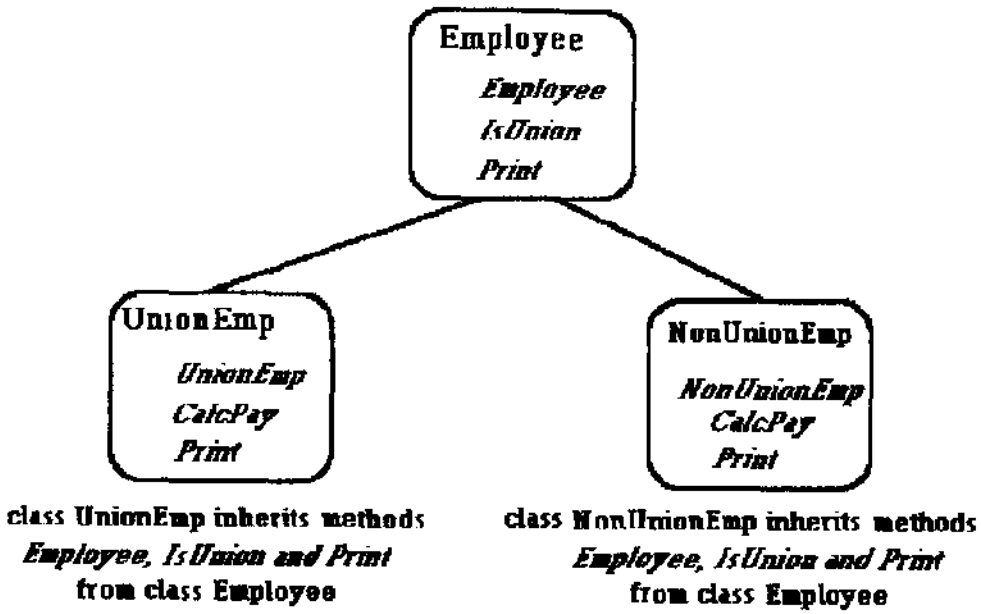
(3) Fewer lines of code to maintain since the classes that were developed contain only the data and methods that make the derived classes different from their root classes.

In the Employee example of inheritance in FIGURE 2.4 of chapter 2, the derived class **UnionEmp** was created from the root class **Employee**. As you can see in FIGURE 8.1, union employees are a special type of employee with only a few differences that make it unique from other types of employees. The differences between the class

Employee and the class UnionEmp are the following:

(1) A formula to calculate the weeks pay.
(2) The union number and OTPAY for union members must be printed on the report.

FIGURE 8.1 - Class Employee, UnionEmp, and NonUnionEmp



A class was defined to relate union employees from the root class Employee. Class UnionEmp was able to inherit the methods *IsUnion* and *Print* from class Employee. In addition to the print method that it inherited, class UnionEmp will have its own print function. Class UnionEmp has access to the salary and hours worked for the union employee. Since these two fields are not accessible to the class Employee they must be printed from the class UnionEmp. Since union employee's weekly pay is calculated differently than other employee's weekly pay, the class UnionEmp should also have its own method to calculate pay. This example has only a few functions to help make it

easier to follow and understand. There could have very easily been several methods and data that could have been inherited which could have saved even more coding.

Code reuse through inheritance happens often in OOP since it occurs each time a derived class is created. The language feature of inheritance lets code reuse happen naturally without much effort on the part of the programmer. Since the development programmer will benefit from code reuse and since it happens naturally from using the language feature of inheritance, code reuse is very common in OOP.

TEMPLATES

A function usually contains code to perform an action on one specific type of data. A function template can be defined that will let one set of code work for multiple types of data. Without templates, if the same action needed to be performed on different types of data, a separate function would have to be written for each data type. Templates allow the programmer to design a generic function that can perform a specific action on any type of data. A parameter is passed to the function that defines the type of data that the function should process. Stuart Johnston writes about Unix Systems Laboratories:[43]

> "[Template] enhances one of object oriented programming's most touted advantages over traditional languages - greater code reusability ... Adding an even higher degree of code reusability results in fewer errors and lower development, testing, and maintenance costs, the company said."

---

43 Stuart J. Johnston, "C++ 3.0 'templates' give greater code reusability", INFO World, 14 October 1991.

The benefits of using function templates are as follows:

(1) Less coding is required since only one function must be written to perform the same action regardless of the data type.

(2) Less testing is required since the template would have already been tested with other data.

(3) Fewer lines of code must be maintained since a single function can be used for multiple type of data.

FIGURE 2.8 shows a function template called "min" that finds the minimum value in an array of numbers, regardless of the type of numbers that are in the array. FIGURE 2.6 performs the same task as found in the template of FIGURE 2.8, except that FIGURE 2.6 can only process an integer array. FIGURE 2.7 performs the same task as found in the template of FIGURE 2.8, except that FIGURE 2.7 can only process a real array. By using the template and having only one function instead of two, the programmer has 50% fewer lines of code to maintain. This has the advantage of the same logic being located in one place.

Both inheritance and templates are language features available in OOP that allow a programmer to reuse code. Inheritance has been available to OOP since the creation of the language style and has been successful at helping programmers to reuse code. Templates on the other hand are fairly new to some OOP languages like C++ and have not been widely used in the real world. Trade magazines[44] and reference manuals feel that templates will soon be a big help to the industry in developing reusable code.

---

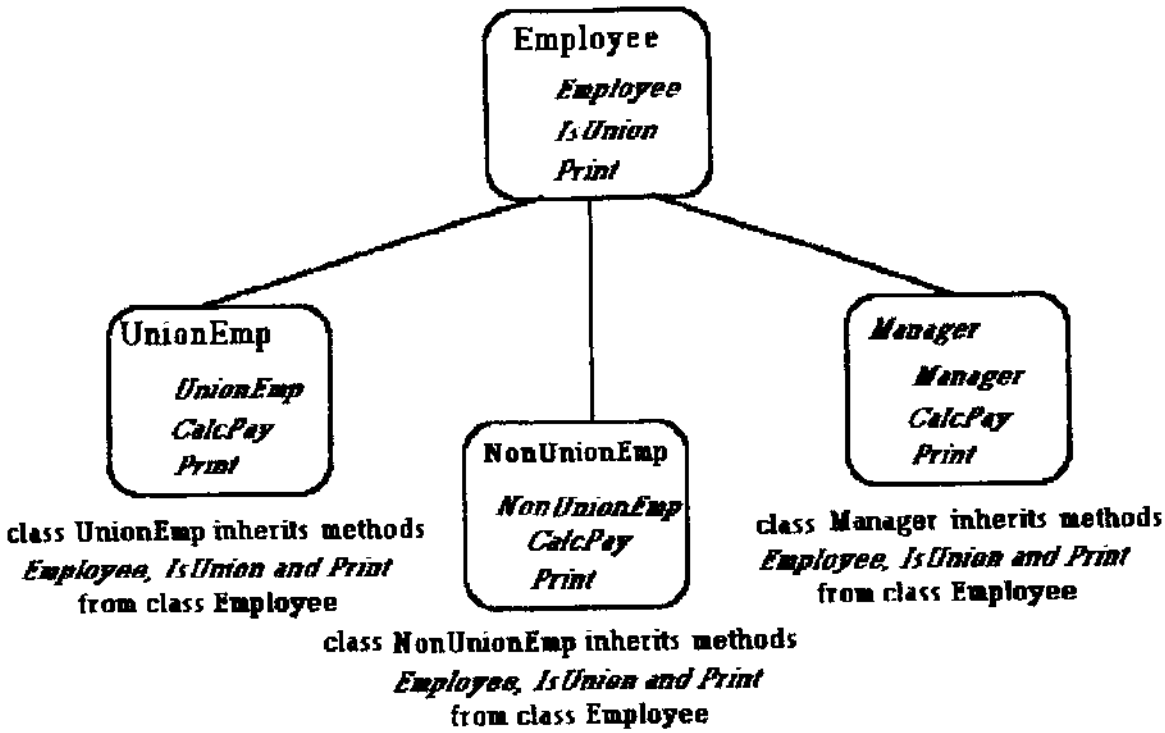
44 Stuart J. Johnston "C++ 3.0 'templates' give greater code reusability", INFO WORLD, 14 October 1991.

MAKING MINOR CHANGES TO REUSABLE CODE.

If a programmer was able to find a class that was almost exactly the same as what is needed, the code from that class can be easily reused. Through the use of inheritance, the derived class can change or add data and methods as necessary that cause this class to be different from its root class. No change would have to be made to the root class so there would be no impact on other classes or programs. As an example, lets assume that a programmer was contacted to say that management employees would no longer be paid overtime. No matter how many hours they work, a manager will always receive pay equal to his or her weekly salary. In this case none of the existing classes of **Employee**, **UnionEmp** or **NonUnionEmp** exactly fit the programmers needs. A

new derived class must be created for managers. The derived class Manager can be seen in FIGURE 8.2 below.

FIGURE 8.2 - Class **UnionEmp**, **NonUnionEmp**, and **Manager** inheritance from class **Employee**



To reuse existing code and only define the code that makes this function different, a new class will be written called **Manager** and it will be derived from the class **Employee** as seen in FIGURE 8.3 below.

FIGURE 8.3 - Sub-class of Manager

```
class Manager : Employee {       // Class Manager definition
private:                         // Accessible only to this classes
  double salary;                 // holds salary for a Manager employee
public:                          // methods accessible by all classes
/*  If 3 string and 2 numeric parameters are passed to this constructor, */
/*  the private field will be set equal to the numeric value passed */
Manager (char LastName[20], char FirstName[15],
            char JobCode[5], long ssn, double sal) :
        Employee (LastName, FirstName,
                        JobCode, ssn);
    {   salary = sal;       }

double Calcpay()
    {   totpay = salary;  }

void Print()
    {
    Employee :: Print();
    cout << salary << "  " << hours << " " << totpay << "\n";
    }
```

The programmer was able to quickly define a slightly modified version of the class **Employee** to meet the new requirement. Those programmers using the class **Employee** would not be impacted by this change. Some programs may need modification so that they can now use the class **Manager** where they were previously using the class **NonUnionEmp**.

SUMMARY

Inheritance is a very common OOP language feature that successfully promotes code reuse. Nearly every object oriented programmer has used this language feature. C++ has a new language feature called templates that was designed specifically for the purpose of promoting code reuse. C++ will soon have a positive impact on the code reuse.

CHAPTER 9

CONCLUSIONS AND RECOMMENDATIONS

Everyone should agree that maintenance costs are currently very much out of hand and that something must be done to try and control these costs. Most POP departments are trying to reduce the cost of program maintenance by purchasing products to use in support of their POP language rather than considering an alternative programming style.[45] Some programmers will argue that they are able to use a procedural programming language and still apply the concepts of object oriented programming.[46] It is possible to simulate some of these concepts but usually not efficiently or easily. Bjorne Stroustrup addresses the concept of simulating OOP by using a language style other than object oriented languages.[47]

> "There is an important distinction here: A language *supports* a programming style if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs: in that case, the language merely *enables* programmers to use the technique. For example, you can write structured programs in Fortran and type-secure programs in C, and you can use data abstraction in Modula-2, but it is unnecessarily hard to do so because those languages do not support those techniques."

---

45 Each POP department that I spoke with, has multiple tools to them to help in reducing the time spent in program maintenance. These POP departments include McDonnell Douglas, Tripos, and JWP Controls

46 One department at McDonnell Douglas is trying to incorporate some of the concepts of OOP into their COBOL programs.

47 Bjarne Stroustrup "What is Object-Oriented Programming?", IEEE Software, May 1988.

From a maintenance standpoint, OOP is better than POP since it requires **fewer function names to remember and coordinate**. A POP language like COBOL, treats its functions as paragraphs or callable sub-programs and each must have unique names. OOP treats its functions as methods within class definitions. One program can reference several different class definitions and each of these class definitions can share the same method name. The OOP capabilities require less work on the part of the programmer and they happen as a natural extension of the language. Therefore, OOP is much better at providing fewer functions to remember and coordinate since the system does much of that work for the programmer.

POP languages such as COBOL, have no features or techniques to help prevent **accidental modifications**. OOP helps prevent accidental modifications by encapsulating classes to limit access to data and methods. Accidental modifications are not totally eliminated with OOP but they can be significantly reduced. Therefore, OOP is a better programming style to prevent accidental modifications.

COBOL allows all procedures within the program to access any of the data in the program. This makes **error detection** difficult. In OOP, a language feature called encapsulation can be applied to a class to help in error detection. This makes OOP a superior language style in helping the programmer detect errors.

The POP language of COBOL has no specific language feature to help limit the impact of a program change. Encapsulation, the same OOP features that helps detect errors can also help limit the impact of a change to a program. Therefore, OOP limits the impact of changes to a program where POP is not able.

POP languages provide the ability to copy source code from a library into a program. The programmer must reuse code from the entire copied program and cannot selectively choose the parts needed or modify the program for only his application. In OOP, the programmer can pick and choose which data and methods to reuse from existing classes. OOP is therefore a better language style at allowing the programmer to create reusable code.

If a COBOL procedural programmer finds code that acts very similiar to a function that he needs to write, the programmer can (1) add the code so that it will work for the existing and new application or (2) copy the existing code and make slight modifications to create new separate code. If the object oriented programmer finds a class that is very similar to what is needed, a derived class can be created that contains only the code that causes the new class to be different from the existing class. If code reuse only happens when someone remembers to apply a technique, such as in COBOL, code will only occasionally be reused. If however, code reuse happens as a result of using a language feature, such as in OOP, code reuse will happen more naturally in the design process.

In conclusion, definite maintenance and code reuse improvements can be realized by using OOP rather than using POP. OOP is found to be the next logical step for programming improvements beyond those improvements experienced from using structured programming.[48]

This thesis supports the hypothesis that OOP is more successful at reducing maintenance costs and improving code reuse than POP. It is therefore my recommendation that programming shops currently using POP languages and experiencing high maintenance and development costs seriously consider switching to an OOP language. Some programming shops have a considerable amount of programs and personnel trained in a specific language and may not be ready to throw that investment away and start from scratch with a new language. In some cases it may be better to use an object oriented version of the currently used procedural language. This may mean switching from C to C++, from LISP to FLAVOR or from COBOL to an object oriented version of COBOL. Object oriented COBOL is currently being developed by an international CODASYL task force.[49]

---

[48]Dick Pountain, "Object-Oriented Programming", BYTE, February 1990, page 157.

[49]Doris Appleby, "COBOL", BYTE, October 1991, page 130.

SELECTED BIBLIOGRAPHY

Appleby, Doris, "COBOL", BYTE, October 1991, page 130.

Arthur, Lowell Jay, Software Evolution the Software Maintenance Challenge, A Wiley-Interscience Publication, 1988.

Atwood, Thomas, "Applying the Object Paradigm to Databases," Computer Language, September 1990. 36.

Bertino, Elisa, and Lorenzo Martino, "Object-oriented database management systems : concepts and issues," Computer, April 1991, 33.

Biggerstaff, Ted J., Alan J. Perlis, Software Reusability, Volume I Concepts and Models, ACM Press, New York, 1989.

Biggerstaff, Ted J., Alan J. Perlis, Software Reusability, Volume II Applications and Experience, ACM Press, New York, 1989.

Bobrow, Daniel G., "The Object of Desire", DATAMATION, May 1, 1989, 37.

Borland C++, Borland International, INC., California, 1991

Buckler, Grant, "OOP is more than a buzzword", Computer Select, June 20, 1991, 31.

Butler, Martin and Robin Bloor, "Object Orientation," DBMS July 1991, page 17.

Caldwell, Tom, "Putting Effective Maintenance into Practice," Computing Canada, October 25, 1990, 44.

Drotos, Diane and Stella Skerlec, "Creating a Rewarding Maintenance Environment," Computing Canada, October 25, 1990, page 42.

Duncan, Ray, "Power Programming, Redefining the Programming Paradigm: The Move Toward OOPLS", PC Magazine, November 13, 1990, 526.

Duntemann, Jeffrey, "OOP: a new perspective on code and data", PC Week, November 14, 1988, 69.

Gorlen, Keith E., Sanford M. Orlow and Perry S. Plexico, Data Abstraction and Object-Oriented Programming in C++, John Wiley & Son LTD., 1990.

Horowitz, Ellis, Fundamentals of Programming Languages, Computer Science Press, 1983.

Hu, David, OO Environment in C++, MIS Press, Inc., 1990.

Johnston, Stuart J., "OOP hailed as way of the future," InfoWorld, May 15, 1989, 17.

Johnston, Stuart J., "C++ 3.0 'templates' give greater code reusability," InfoWorld, October 14, 1991.

Lippman, Stanley B., C++ Primer, Addison-Wesley Publishing Company, Massachusetts, 1991.

Marrs, Keith, "Object-Oriented Database Management Systems: The State of the Art", McDonnell Douglas Corporation Report B1659, 1989.

Martin, James, "OOP goes beyond the commonsense meaning of 'object'," PC Week, September 11, 1989, 76.

Mullin, Mark, Object Oriented Program Design with Examples in C++, Addison-Wesley Publishing Company, Massachusetts, 1989.

"Objects at Large", Release 1.0, September 19, 1990, page 4.

Parikh, Girish, Techniques of Program and System Maintenance, QED Information Sciences INC., Massachusetts, 1988.

Peterson, Robert, "Object-Oriented Programming", BYTE, February 1990, page 257.

Pountain, Dick, "Object-Oriented Data Base Design", AI Expert, March 1987, 26.

Pratt, Terrence W., Programming Languages Design and Implementation, Prentice Hall INC., 1984.

Sharon, William David, "Comparing Object-Oriented and Structured Methods," presented at Showcase VI, September 25, 1991, St. Louis, MO.

Smith, Jerry D., Reusability & Software Construction C & C++, John Wiley & Son , 1990.

Snyder, Alan, "Encapsulation and Inheritance in Object-Oriented Programming Languages," OOPSLA '86 Proceedings, September 1986, 38.

Stroustrup, Bjarne, The C++ Programming Language, 2nd edition, Addison Wesley Publishing Company, Massachusetts.

Stroustrup, Bjarne, "What is Object-Oriented Programming?" IEEE, May 1988.

Tracz, Will, Tutorial: Software Reuse: Emerging Technology, IEEE The Computer Society of the IEEE, 1988.

Urlocker, Zack, "Teaching object-oriented programming," Journal of Object-Oriented Programming, July-August 1989, page 45.

Van Genuchten, Michiel, "Why is software late?," IEEE Transactions on Software Engineering, July 1991, page 582.

Verity, John W. and Evan I. Schwartz, "Software Made Simple," Business Week, September 30, 1991, page 94.

Weiner, Richard S., and Lewis J. Pinson, An Introduction to Object-Oriented Programming and C++, Addison-Wesley Publishing Company, Massachusetts, 1988.