INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

PREPROCESSOR FOR C++ CLASS IMPLEMENTATION

Zahra Djalalian

A Thesis

In

The Department

Of

Computer Science

Presented In Partial Fulfillment Of The Requirements

For The Degree Of Master Of Computer Science

Concordia University

Montreal, Quebec, Canada

March 2000

Abstract

Preprocessor for C++ Class Implementation

Zahra Djalalian

A single file approach for class implementation in C++ programming language is addressed in this thesis. The objective of this thesis is to find a simple and efficient class implementation approach for C++ programming language. The current C++ class implementation named as "two files approach" has many drawbacks and also has some advantages. This thesis develops a new approach, the single file approach, which keeps the two files approach's advantages and eliminates its disadvantages. In the single file approach the developer puts all class information into one file and thus it is easier for the developer to develop only one file (canonical file) for class information. In this thesis a preprocessor program is developed to generate four different files from the canonical file, i.e. the header file and the implementation file for the compiler, and two documentation (interfaces) files for clients. These output files do not have the interface and implementation files' drawbacks of the two files approach. Also, in the single file approach the separation of class information is done automatically which saves the developer's time.

Acknowledgments

I express my resounding gratitude to my thesis supervisor, Prof. Peter Grogono, who was able to reveal the wonder and excitement which accompany the first insights into a new problem; always challenged and motivated me to produce the best in my work. His invaluable suggestions and generous support through the course of this research at Concordia University is highly appreciated.

I would like to thank Mrs. Halina Monkiewicz for her great help and advises.

It is very difficult for me to find some words to express my feelings toward my husband and best friend, Dr. Aghil Yousefi-Koma. The success of this degree would not have been possible without his continuous support and encouragement. I would like to express my deepest appreciation for his patience and support during these two years. He managed our student life very well and always as a teacher guided me during my graduate studies. I would also like to thank my children, AmirHossein, Hannaneh, and AmirAli for understanding their student mother.

I would like to give my heartfelt thanks to my parents, my brother and my sisters for their help, encouragement, and prayers. I would like to appreciate my parents-in-law for their sincere support through the course of my study.

Above all, my deepest appreciation to Allah almighty who gave me the strength and opportunity to rise to this stage. I wish for his mercy for the rest of my life.

Dedicated to

My husband, Dr. Aghil Yousefi-Koma; my kids, AmirHossein, Hannaneh, and AmirAli

my parents, Seyyed Hassan and Ozra,

my parents-in-law, Mohammad Hossein and Kafieh

Table of Contents

1 Introduction **1**

1.1 Modular Programming 2

1.2 Information Hiding 3

1.3 Abstract Data Type 4

1.4 Class 5

1.5 Interface and Implementation Files 6

1.6 Advantages of Two Separate Files 7

1.7 Overview of the Thesis 8

1.8 Thesis Outline 10

2 Module Specification **12**

2.1 Object Oriented Approach in Eiffel 12

2.1.1 Disadvantages of Object Oriented Approach in Eiffel 15

2.2 Object Oriented Approach in Java 17

2.2.1 Disadvantages of Java Classes Definition 18

2.3 Object Oriented Approach in Smalltalk 18

2.3.1 Internal and External Views of an Object 20

2.4 Object Oriented Approach in Ada 21

2.4.1 Disadvantages of Ada Package Implementation 22

2.5 Object Oriented Approach in Modula-2 24

2.5.1 Disadvantages of Module Implementation in Modula-2 24

2.6 Object Oriented Approach in C++ 25

2.6.1 Disadvantages of Class Implementation in C++ 25

3 Comparison of Two Approaches **27**

3.1 Single File Approach 27

3.2 Single File Approach vs. Two Files Approach 29

3.2.1 Need for Interface and Implementation Files. 31

3.2.2 Duplication 31

3.2.3 Declaration and Use 32

3.2.4 Interface File Contains Implementation Information 33

3.2.5 Interface File Contains Information for Compiler 33

3.2.6 Encapsulation 34

3.2.7 Missing Information 35

3.2.8 Comments are Big Problem in Interface File 35

3.2.9 Inline Functions 36

3.2.10 Grouping Definitions 36

3.2.11 Clients Recompilation 37

3.2.12 Error Handling 38

3.3 Conclusion 38

4 Usable Single File Approach for C++ Compiler **39**

4.1 Single File Approach Implementation for C++ 40

4.2 Canonical File Grammar 41

4.3 Parser and File Generator 47

5 Implementation of Preprocessor and Code Generator Programs **52**

5.1 Input and Output Files 53

5.2 Preprocessor Program Classes 54

5.3 Parser Algorithm 59

5.4 File Generator Algorithm 62

6 Discussion **65**

7 Conclusion **70**

7.1 Suggestions for Future Extension of this Research 72

References **73**

Appendix A **75**

Preproc.cpp Program file 76

Commentc.h file 79

Feature.h file 80

Filename.h file 82

Library.h file 83

List.h file 84

Module.h file 85

Node.h file 86

Parameter.h file 87

Supplier.h file 88

Token.h file 89

Canonical file example 91

Interface file example 93

Interface2 file example 94

Header file example 95

Implementation file example 96

List of Listings

Listing 2.1 An example of a class PUBLICATION definition in Eiffel 14

Listing 2.2 A derived class JOURNAL definition in Eiffel 15

Listing 2.3 A sample of Java class definition 19

Listing 2.4 C++ interface file 19

Listing 2.5 C++ implementation file 19

Listing 2.6 A package interface in Ada 23

Listing 2.7 A package body in Ada 23

Listing 3.1 An example of C++ class interface file (counter.h) 30

Listing 3.2 An example of C++ class implementation file (counter.cpp) 30

Listing 4.1 The canonical file grammar 42

Listing 4.2 A sample of a canonical file 48

Listing 4.3 Implementation file: output of the file generator 50

Listing 4.4 Header file: output of the file generator 51

Listing 4.5 Documentation (interface) file: output of the file generator 51

Listing 4.6 Documentation2 (interface2) file: output of the file generator 51

List of Figures

Figure 4.1 The output of the parser, i.e. a tree 48

Figure 4.2 The #include subbranch 49

Chapter 1

Introduction

Today, there are many large programs in the computer world which contain from tens of thousands up to millions of lines of codes. The development of large programs differs from the development of small programs in several ways. First, large programs should be developed by a team of programmers rather than by a single individual. One or more teams of programmers are required, and there has to be co-operation both within each team and among different teams. Second, a large program itself cannot be written as a single file, but has to be split up into several files. For this approach to be possible there should be some mechanisms for dividing the program into separate parts which only interact with one another through small strictly defined interfaces.

Large systems, by their nature, are usually in use for many years and during this time they may undergo substantial modifications. Some of the changes will be due to the discovery of errors, but others will be due to changes in the system requirements, which could not be foreseen when the original system was designed. As it is not practicable to redesign large systems from scratch, due to their high cost and the time involved, the systems should be designed and written so that they can be easily modified. The best way

to achieve this is for the programs to be written in such a way that the effect of any change is localized (Wilson and Clark, 1993).

For this reason large programs or software systems are usually broken down into small parts. Each part of a large program has specific *attributes* (attribute means a characteristic quality of a component) and can be developed and tested by a programmer(s) (also called *developer*(s)). A part of a large program should be a coherent unit and is called a software component. Therefore, the program has to be split into a number of components. We propose that software components should consist of exactly one file and can be coded and tested individually.

Each software system consists of several small components. In this way, a library of software components can be made available "off the shelf" for the use in the construction of new programs. After all software components are coded and tested by developers individually, the software manager uses them and put them together in a package to produce a software system.

1.1 Modular Programming

Modular programming is simply the decomposition of a software system into separate, discrete parts. It is an intuitive process to break a large complex system problem down into a manageable number of components. The program should be written so that it can be easily reusable in other systems and can be extended in future. To design and write large programs, developers should build them up from small components. Each component should be designed in such a way that it can be used in more than one program. If components are designed in this way, they will be reusable.

In modular programming languages each software component is called a *module*. Each module has some specific attributes and is able to perform certain functions. A module is an arbitrary collection of types, attributes (data) and functions. A module can export any combination of its type, attributes and functions (i.e., make them available to other modules) and can import types, attributes and functions from other modules.

Modules allow a large system to be designed by specifying what each module is to do and what its interface with the rest of the system is to be. Once this is done, different developers may implement each module without any need for consultation. Modules can be joined together to form complete programs.

Modularity is a key to achieve the aims of reusability (*reusability* means that a program or a part of a program can be used for another system) and extendibility (if a program or a part of a program can easily be extended by adding new functions, it has *extendibility*). Modularity helps designing a software system to be both extendable and reusable. Also, it is very easy to change or extend each module without changing other modules.

1.2 Information Hiding

Each module is written by an individual programmer or by a team of programmers. The term *developer* is used to represent the programmer or programmers who write the module. After the module has been coded, the developer prefers to protect it from other developers or clients who use this module. In large programs, it is often desirable for information to be visible to several program modules and yet to be hidden from the rest of the program modules. In modular programming, it is possible to decide

what information is to be visible from outside the collection and what information is to be kept hidden. This can be achieved in various ways.

One method is to divide a module into two parts. Every module has an interface that says what it does and an implementation that says how it works. More formally, the interface describes the desired behavior and the implementation ensures that the desired behavior is actually achieved. The part that tells the client "what" the module does acts as the interface with the client. While the part containing the details of "how" the operations are carried out is hidden from the client. Good programming practice requires that the implementation is "hidden": that is, not visible to clients. The principle of information hiding was proposed by Parnas (1972).

Some module information can be hidden from the module clients by putting them in the private part of the module. This means that no information in the private part is available to the clients of the module.

1.3 Abstract Data Type

An *abstract data type* (ADT) is a particular kind of module. An ADT is a data type whose domain and operations are specified independently from any particular implementation. It has one principal attribute (a data structure) and export functions that manipulate this structure. The structure itself is not exported (i.e., it is hidden).

In the world of software design, it is now recognized that abstraction is an absolute necessity for managing immense, complex software projects. It is hard to design, understand, or prove the correctness of large software products without using abstraction in various forms. Data abstraction is a powerful technique for reducing the complexity

and increasing the reliability of programs. There are two kinds of ADTs, depending on instances of the data structure, i.e. one instance of the ADT or many instances of the ADT. Object oriented languages support this view by providing a built-in structured type known as a class.

1.4 Class

In object oriented programming languages, a module is called a *class*. A class is a module that exports a single type. The concept "type" and "class" are related but not identical. One common point of view is that a class is an implementation of a type. Clients of a class can declare as many instances of a type as they need. Classes can be used to implement an ADT, but a class is not necessarily an implementation of an ADT.

Object oriented programming encourages developers to reuse existing code rather than rewrite functions from scratch. Classes are easy to reuse and extend, and object oriented programming code tends to evolve from existing classes the way trees grow by extending their branches.

A class has two components: data member and member functions. Data members of a class are its attributes. Member functions of a class manipulate its data members. Data members and member functions can be declared as private or public.

Clients of a class can access to the public class members directly. Making data member public allows the client classes to inspect them directly. Class members that are declared private are considered inaccessible information to the client. Private class members can be accessed only by the class member functions.

The class developer is free to choose which member is private and which one is

public. In most classes the private part contains data, and the public part contains the functions which manipulate the data members and/or data members. Since only the classes member functions can access the private data, the class developer can offer a reliable product, knowing that external access to the private data is impossible.

1.5 Interface and Implementation Files

In some object oriented languages a class is coded in two separate files: an interface file and an implementation file. The interface file describes the behavior of the data type without reference to its implementation. The interface file presents the class's public interface to the user in the form of function prototypes. The implementation file creates an abstraction barrier by hiding the concrete data representation as well as the code for the operations. The implementation file also provides function definitions for all member functions.

In normal use, C++ packages the class declaration and the class definition into separate files. The interface file, which is a header (.h) file, contains the class declaration and the implementation file includes the function definitions for the class member functions. The clients of the class can see only its declaration, which is in the interface file. In fact, the interface file declares what the class does, which is important for clients to know. The implementation file defines how the functions are implemented, and it is not important for clients.

The interface file is accessible by all clients, and clients write the interface file name in their file to access to the class functions. When a client program executes, the client object uses the implementation of the server object to execute its functions.

Eventually, one class can be a client of another class. In well-written object-oriented programs, the main program simply creates the principal objects and then sits back and lets them communicate with each other.

1.6 Advantages of Two Separate Files

One of the advantages of using the class in C++ is that each class has two files, the interface file and the implementation file. The interface file is what the class owner "publishes" and the implementation file is the class owner's "secret". The interface file introduces class member functions and data members. The class developer defines all member functions in the implementation file. The class developer or maintainer is free to change the implementation file and then if the interface file stays the same, client classes do not need to be recompiled.

The important parts of a class are its member functions. A client who wants to use a class must know its member functions. Thus, the class public member functions need to be shown to the client while some information like private member functions and all member functions declarations and data members should be kept out of the client access. C++ language puts some information in the interface file, which is necessary for the client who uses the class to understand what the class does, and for information hiding it puts other information in the implementation file, which is not accessible by the client.

To protect classes from damaging by the client, the class's implementation files, which consists of functions definitions and some private data members, are not accessible by the client. The interface file in C++ must contain declarations of all private and public members (function and data) even though the client cannot use them. Consequently, the

class developer who changes private data may cause all clients to be recompiled.

Whenever the client needs to use this class, the name of class interface file should be written in the top of the client main program. Then, the C++ compiler uses the class implementation file to access the class functions definitions and other private data members, and runs it. Hence, the class owner is sure that nobody can destroy the class information.

1.7 Overview of the Thesis

This thesis concerns the implementation of classes in C++ language. Object oriented programming languages like C++ split up the program into small parts. C++ language calls each part as a *class*. Class programmers prefer to hide the class definition information from the clients of the class. C++ keeps class information in two files. Class declaration has to be put into an "interface" file and class definition has to be kept in an "implementation" file.

Whether C++ language is successful to separate the class information into two files to achieve information hiding in a perfect way or not, is discussed in this thesis. The current approach consists of two files, i.e. the interface and implementation files for class declaration and definition, named as Two Files Approach. Two files approach has some strengths and weaknesses.

On the good side:

- There is partial separation of "interface" (the interface file) and "implementation" (the implementation file).
- The implementation file can be changed and recompiled without recompiling other

files. although it may be necessary to relink the program.

These features are offset by some disadvantages:

- Although C++ has two kinds of files (.h and .c or .cpp), in fact these are not interface and implementation files. The "interface" file contains some of the "implementation" file information. The interface file has also information about data members and private member functions.
- The developer must write each function prototype twice, once in the interface file and once again in the implementation file. This is particularly annoying when a function must be changed (e.g., adding a parameter) because the same change must be made in two places. Moreover, if the changes are not made accurately, the compiler diagnostics may be misleading because C++ allows overloading.
- Information is missing from the implementation file. For instance, when one is reading an implementation file, one cannot tell whether a function is public or private. The same problem applies to other features such as the "virtual" attribute.
- The interface file serves two purposes: the compiler reads it to obtain the information about the declared class; and clients read it to know how to use the class. Consequently, it contains information needed by the compiler but not by the programmer (e.g., the private parts of the class). Moreover, interface files comprises other information needed by the programmer but not by the compiler (e.g., comments).

The separation of interface and implementation files is important. Although the interface and the implementation files must be kept separate, the developer does not necessarily need to write them separately. They can be written as one unit and be separated later. One possibility would be to provide four files: a compiler interface, two

human-readable interfaces, and an implementation. But this would make the problem worse!

The goal of this thesis is to eliminate the disadvantages of the two files approach without losing its advantages. Developing a Single File Approach in this thesis, a preprocessor program is designed and implemented to overcome the two files approach drawbacks while keeping its advantages. In the single file approach, the developer can implement all class information in one file named as a *Canonical file*. Then, a program named as preprocessor uses this canonical file as an input file to generate four different files for different purposes, i.e. implementation and header files for the compiler, interfaces files for class client.

1.8 Thesis Outline

An introduction to the thesis is provided in Chapter 1. The problem definition and a solution to this problem are introduced. The advantages and disadvantages of the two files approach, are discussed, and a new approach, i.e. single file approach, developed in this thesis is proposed to eliminate these disadvantages.

A literature survey on class implementation in different object oriented programming languages is presented in Chapter 2.

Comparison of two approaches (single file and two files approaches) is illustrated in Chapter 3. Each approach has some advantages and also drawbacks. In Chapter 3, it is explained that which approach is better and why.

Respecting to the results of Chapter 3, Chapter 4 shows how the two files approach in C++ language can be modified. The program named as preprocessor, which

is developed in this thesis is introduced.

Chapter 5 illustrates the details of how the preprocessor program is implemented. It is shown what the input and outputs of the program are and how the program works.

The discussion about the preprocessor program and how useful they are is presented in Chapter 6. It is also shown that the C++ class developers can employ the preprocessor program through a single file approach to obtain a much easier and faster class implementation.

Chapter 7 presents the conclusion, a brief list of the research contributions of the thesis and suggestions for the future work in this area.

Chapter 2

Module Specification

Object oriented programming languages employ module specification and almost all of them provide information hiding, but in different ways. In this chapter some of them are studied to show how they define and implement a module and what the advantages and disadvantages of each approach are.

2.1 Object Oriented Approach in Eiffel

The aim of Eiffel is to produce an object-oriented system, which is simple and efficient and can be used in the production of high quality and reliable production-level software (Wilson and Clark, 1993; Meyer 1992). Classes are the components used to build Eiffel software. A class describes the properties of a set of possible data structures, or objects, which may be created during the execution of a system that includes the class; these objects are called the *instances* of the class. Eiffel also makes a clear distinction between a class and an object. Classes in Eiffel correspond to types in a traditional language while an object is an instance of a class and is produced during the run time. In

Eiffel all objects are accessible indirectly through a reference, i.e. a pointer.

The text of an Eiffel program consists of a set of class definitions. At run time objects are created dynamically. A class introduces a set of features. Some features, called *attributes*, represent fields of the class's direct instance; others, called *routines*, represent computations applicable to those instances. All actions involve performing an operation on some objects, which can cause other objects to be created or the attributes of objects to be changed. Listing 2.1 demonstrates an example of a class PUBLICATION definition in Eiffel.

Instead of having public, private and protected members (known as *features* in Eiffel), features, which are to be visible to the client of a class must be explicitly exported. As mentioned earlier there are two kinds of features in Eiffel: data items, i.e. *attributes*, and *routines*.

In Eiffel, an attribute of an object may be made read only to the client. Allowing the representation of an object only to be accessed by a client through a procedural interface has much to recommend it. The decision to allow Eiffel attributes to be visible is made purely on efficiency grounds, that is, to save the overhead of a routine call.

The class JOURNAL is given as Listing 2.2, which is a derived class of the class PUBLICATION (Listing 2.1). In Eiffel terminology, the class JOURNAL is a *descendant* of the class PUBLICATION, which is known as the parent class. Eiffel does not have the scope resolution operator of C++ and so identifiers have to be renamed in a descendant class if there is any name clash. Since a constructor is always called Create, to use the constructor of the class PUBLICATION in the class JOURNAL, the constructor must be renamed.

Listing 2.1 An example of a class PUBLICATION definition in Eiffel.

```
class  PUBLICATION

        export
                title, catalog_no, publisher, do_print, cont_print

        feature
                the_title, the_cat_no, the_pub: STRING;

        title : STRING is
        do
                result := the_title.dublicate
        end; --title

        catalog_no : STRING is
        do
                result := the_cat_no.duplicate
        end; --catalog_no

        publisher : STRING is
        do
                result := publisher.duplicate
        end; --publisher

        do_print is
        do
                io.putstring(the_title);
                io.putstring("\n")
        end; --do_print

        cont_print is
        do
                io.putstring("Contents are :");
                do_print
        end; --cont_print

        Create(ti, cat_no, pub : STRING) is
        do
                the_title := ti.duplicate;
                the_cat_no := cat_no.duplicate;
                the_pub := pub.duplicate
        end--Create

end--class PUBLICATION
```

2.1.1 Disadvantages of Object Oriented Approach in Eiffel

The purpose of *repeat* PUBLICATION in the export list of the class JOURNAL is to state that all items exported from the class PUBLICATION are also to be exported from the class JOURNAL. Although the export list restricts the features, which are visible to the clients of a class, all features of a parent class are automatically available to its descendants. This means it is possible for a descendant to export features, which are not exported by its parent class.

Listing 2.2 A derived class JOURNAL definition in Eiffel.

```
class JOURNAL
        export
                volume, part_no, repeat PUBLICATION
        inherit
                PUBLICATION rename Create as Publication_Create
                redefine do_print
        feature
                the_vol, the_part : INTEGER;
        volume : INTEGER is
        do
                result := the_vol
        end; --volume

        part_no : INTEGER is
        do
                result := the_part
        end; --part_no

        do_print is
        do
                io.putstring(the_title);
                io.putstring(",");
                io.putint(the_vol);
                io.putstring("\n")
        end;--do_print
        Create(ti, cat_no, pub : STRING; vol, part : INTEGER) is
        do
                publication_Create(ti, cat_no, pub);
                the_vol := vol;
                the_part := part
        end--Create
end--class JOURNAL
```

Inheritance is about making information available to descendant classes so that software can be reused. It therefore conflicts, to some extent, with the concept of information hiding which is about restricting access. Assume the feature the_title is not exported by the class PUBLICATION thus, it cannot be used by the client of PUBLICATION. However, it is available to the descendant class JOURNAL. Also, if it is added to the export list of JOURNAL, it becomes available to clients of JOURNAL. Hence, the Eiffel language can not achieve information hiding effectively.

The drawback of inheritance in Eiffel is that the number of identifiers which are in scope can become very large and since a parent class is typically descended from its own parent, it can be difficult to determine where exactly an identifier has been declared.

Another disadvantage of the Eiffel language is to provide information hiding. Information hiding means that how the author of a class can provide authors of client classes with a clear description of the facilities offered, while developer is keeping some features inaccessible. The full class text is not appropriate for this purpose, as it contains implementation details as well as interface information, which is in contradiction with the principle of information hiding. The designer of a parent class may decide that a certain feature is to be hidden from its clients. For instance, it is concerned with implementation details. Allowing a feature hidden in a parent class to be exported to the clients of a descendant is against the idea of data encapsulation and information hiding.

The Eiffel approach for problems of this nature is to provide software tools. The proper description is provided by a *short form* of a class, also called an *abstract form*. A short form is a text, which has the same structure as the class but does not include non-public elements. The short form should be used as interface documentation for the class.

The responsibility for producing such short forms should not lie on developers or clients but on automatic tools of the Eiffel environment. For instance, the effect of applying the tool *short* to a class is to expand all inherited features to give one flattened class. The class in which each inherited feature is declared is given as a comment. There can also be a conflict between the aims of inheritance and information hiding in Eiffel class implementation (Wilson and Clark, 1993; Thomas and Weedon, 1995).

The short form includes information about the immediate features of a class. To obtain more complete information such as treating all features and inherited information, there is another form named *flat-short form*. The flat-short form is similar to the short form, but it does not show the original classes of derived features (Meyer, 1992).

2.2 Object Oriented Approach in Java

Java is a pure object oriented language, thus everything must be inside an object and everything is descended from a single object class. Classes contain data members (data type declaration) and methods (functions which operate on the data members of a class). Class methods contain data type declarations and statements (Arnold and Ken, 1998; Daconta, 1996). An example of a Java class definition is given in Listing 2.3.

As it is seen Java class implementation is similar to that of C++. The big difference between C++ class and Java classes is that C++ defines class in two files as interface and implementation files, while in Java all definition and declaration information is explained and written in only one file. Considering Listing 2.3, the corresponding class in C++ language can be defined in two files (interface and implementation) as Listings 2.4 and 2.5.

2.2.1 Disadvantages of Using Java Classes

In Java, class libraries are referred as packages. Packages are collections of classes, usually grouped according to functionality. The functionality that Java offers in the form of system classes for working with data, graphics, etc., is provided in packages which can be imported into the program.

The disadvantage of using packages is that the developer must write the class information twice, e.g. in the class file and in the package. The class file contains all public and private class member information. To achieve information hiding and inheritance in Java, some of the class information has to be put into the package. Therefore, class information should be written in two different files with two different formats. (Flanagan, 1997; Anuff, 1996)

2.3 Object Oriented Approach in Smalltalk

Smalltalk is a language somewhat is based on a surprisingly small set of concepts; namely objects, messages, classes, subclassing and inheritance (object is a component of the Smalltalk system represented by some private data and a set of methods or operations). An object includes private data or state, which is kept within the object. An object is also capable of computation. It can respond to any predefined set of messages (a synonym for operation, invoked when a message has received by an object). This message set is referred to the message protocol supported by the object. When an object receives a message, first it must check if it understands the message, and if so then, what its response should be. If an object can respond to a message directly, a method or function corresponding to that message is selected and evaluated. Consequently, the

Listing 2.3 A sample of Java class definition.

```
class Entity {
        protected String name;
        protected int x;
        protected int y;
        public Entity()
        {
                x = y = 0;
                name = null;
        }
        public void finalize()
        {
                //code here
        }
        public void setXY(int inX, int inY)
        {
                //code here
        }
}
```

Listing 2.4 C++ interface file.

```
class Entity {
        protected:
                char* name;
                int x, y;
        public:
                Entity();
                ~Entity();
                void setXY(int inX, int inY);
        }
```

Listing 2.5 C++ implementation file.

```
Entity :: Entity()
{
        x = y = 0;
        name = NULL;
}
Entity :: ~Entity()
{
        //code here
}
void Entity :: setXY(int inX, int inY)
{
        //code here
}
```

result of evaluating the method is returned to the sender of the message. Message protocol is a set of messages to which an object can respond (Lalonde and Pugh, 1990).

2.3.1 Internal and External Views of an Object

Objects in Smalltalk encapsulate both functions and data. They support the well-accepted software engineering concept of information hiding to control the complexity of large programs, which need to be partitioned into modules. Moreover, the information should be hidden as much as possible within a module. Also, the interface presented to clients should be minimized. It is useful to think of a Smalltalk object as an object providing two different views of itself: one for clients of the object and another for developers of the object. These views are called the external and internal views respectively.

The internal view describes the representation of the object and the algorithms, which implement the methods (or functions). The external view is looking at the object as seen by clients. The external view, or what the client can do with an object, is described by its message protocol, i.e. a set of messages to which the object responds. To a client, the internal view of an object is private. It is owned by the object and may not be manipulated by other objects unless the object specifically provides a protocol for doing so.

The separation between the internal and external views of an object is fundamental in the programming philosophy embedded in Smalltalk. To use an object, it is necessary to understand only its protocol or external view. Providing the message protocol or external view is not changed, the fundamental advantage of this approach is

that the internal view may be changed without impacting clients of the object. Similar facilities for information hiding are provided by the module facility in Modula-2 (see section 2.5) and the package in Ada (Schildt, 1987; Volper and Katz, 1990).

2.4 Object Oriented Approach in Ada

Ada offers a modular construction for grouping logically related program elements (Volper and Katz, 1990). An Ada module is called a *package*. Packages are purely syntactic notions and do not have semantic values. They provide ways to distribute system elements into coherent subsystems, but they are only needed for readability and manageability of the software. A package consists of a set of related constants, a library of routines, a set of variables, and an abstract data type implementation. The interface with the outside world is called the package specification. The implementation details are held in a separate package body. Package specification and definition modules should always be as small as possible, since they form the interfaces among what are otherwise self-contained units. A large interface is often a sign that the decomposition of the problem could be improved. An identifier declared in a package specification is visible from outside the package. Clients of the module can declare variables and call the functions. The only information about functions, which is given in the package specification, is that they are needed by the client to call the function; namely, the number and type of the parameters and the type of the returned function values. Details of the implementation of a function (a function body) are hidden in the package body, since there is no need for a client to have access to this information (Watt, Wichmann, and Findlay, 1987).

Information hiding is supported in Ada by the two-tier declaration of packages. Every package comes in two parts called "specification" or "interface" and "body". The interface lists the public properties of the package as exported variables, constants, types and functions. Whereas, it only gives the headers, names and types of arguments, and names and types of result variables of functions. The body part of a package provides the functions' implementations, and adds any necessary secret elements.

The keyword package by itself introduces a package interface and a body can be introduced by a package body. For instance, the body of a REAL-STACK package (Listing 2.6) might be declared as Listing 2.7.

2.4.1 Disadvantages of Ada Package Implementation

The disadvantages of Ada package implementation are given as following:

- In order to change the implementation in Ada package implementation both the package specification and the body must be modified.
- Another drawback of Ada package implementation is the existence of detailed information in the interface file. Although the interface file is supposed to function purely for interfacing, all details of STACK representation, as given by the declaration of types STACK and STACK- CONTENTS (Listing 2.6 and 2.7), appear in the interface file.
- In Ada package implementation a package does not by itself define a type, but must separately define a type STACK. As a result of this separation, the developer who builds a package around an abstract data type implementation, needs to create two different names, one for the package and one for the type.

Listing 2.6 A package interface in Ada.

```
package REAL-STACK is
        type STACK-CONTENTS  is array ( POSITIVE range<>) of FLOAT;
        type STACK (capacity: POSITIVE) is
                record
                        implementation: STACK-CONTENTS (1.. capacity);
                        count: NATURAL :=0;
                end record;
        procedure put (x: in FLOAT; s: in out STACK);
        procedure remove (s: in out STACK);
        function item (   s: STACK) return FLOAT;
        function empty(s: STACK) return BOOLEAN;
        overflow, underflow: EXCEPTION;
end REAL-STACK;
```

Listing 2.7 A package body in Ada.

```
package body REAL-STACK is
        procedure put(x: in FLOAT; s: in out REAL-STACK) is
                begin
                        if s.count = s.capacity then
                                raise overflow
                        end if;
                        s.count := s.count +1;
                        s.implementation(count) :=x;
                end put;
        procedure rename(s: in out STACK) is
                        ... implementation of remove ...
                end remove;
        function item(s: STACK) return x is
                        ... implementation of item ...
                end item;
        function empty(s: STACK) return BOOLEAN is
                        ... implementation of empty ...
                end empty;
end REAL-STACK;
```

- Package STACK, fails to implement the principle of information hiding: the declarations of type STACK and STACK-CONTENTS, are in the interface file, allowing clients to access the representation of stacks directly.

- The type STACK must now be declared twice: once in the non-private part of the interface, where it is only specified as private; once again in the private part, where the

full description is given. Without the first declaration, a line in the form of: REAL-STACK will not be legal in a client, since clients have access only to the entities declared in the non-private part (Meyer, 1997).

The disadvantages of the Ada package implementation show that this implementation is not a secure approach.

2.5 Object Oriented Approach in Modula-2

In Modula-2, there are two modules known as the declaration module and the implementation module. All identifiers in a declaration module are automatically available in the corresponding implementation module, and they must be explicitly exported by means of an export list. Also, the contents of the implementation module are hidden and cannot be accessed by clients of the module. Information on how to call a function is given in the declaration part, while the implementation of a function is kept in the implementation part.

A declaration module in Modula-2 may be compiled separately from its implementation module if there is no equivalent of a private part in the declaration module. The only way to export a type, while keeping its structure hidden, is to give the type name in the definition module and use pointers in the implementation module (Schildt, 1987).

2.5.1 Disadvantages of Module Implementation in Modula-2

The main drawback of the module implementation in Modula-2 is the separation of the declaration part and implementation part in two different modules. To write two

different modules is very difficult for the module developer. The module developer should spend too much time and pay attention to separate these information.

2.6 Object Oriented Approach in C++

In C++, the equivalent of the package or module is called a class. A class definition in C++ consists of data members, which define the internal representation details, and member functions through which the internal details can be accessed. Member function bodies can be declared within a class definition, but usually they are declared separately and only the function prototypes are declared in the class definition. Another important feature of C++ programs is their physical decomposition into separate files whose contents may be compiled separately (Ellis and Stroustrup, 1990).

2.6.1 Disadvantages of Class Implementation in C++

When dealing with large systems composed of hundreds of subprograms, it is important to ensure that a change in one module does not necessitate the recompilation of all other modules. In C++ any change in the interface file, even a comment change, causes recompilation of all clients. This may cause a serious problem for programs, which need hours or days to recompile. Many tools, particularly "make" rely only on revision time or time stamp. This means if a ".h" file is changed more recently than a ".c" file, which includes the ".h" file, the ".c" file must be recompiled. This problem can be avoided by using a better criterion than a time stamp (as described in the single file approach, section 3.2.11).

The class developers should spend more time to write two separate files for each

class. They must be very careful to put corresponding class information to each file to achieve information hiding.

The interface file contains information for compiler as well as information for the client. The compiler needs information, which should not be accessible by the client. On the other hand, the compiler must parse comments, which takes too much time. The C++ header files for library files are quite long. (e.g., windows.h for windows NT API has more than 100k lines.) Since these files must be parsed often and this takes time, almost all comments are removed out of them. Consequently, where programmers really need comments, there is not any.

Chapter 3

Comparison of Two Approaches

As mentioned in chapter 2, C++ programming language employs the two files approach for the class implementation. Also, some of disadvantages of this approach were discussed. In this chapter a new approach for C++ class implementation is introduced and then compared with the current approach of C++ class implementation, i.e. two files approach.

3.1 Single File Approach

A new approach for C++ class implementation named as *Single File Approach* is introduced in this section. As illustrated in the previous chapter there are some disadvantages in the two files approach. For this reason a single file approach is developed in this study to overcome these drawbacks. The single file approach has only one file for class declaration and definition. It means instead of having two files (header file and implementation file), there need to be only one file that contains all necessary information for declaring and defining a class. This file is called a *canonical* file.

Since the single file approach needs one file for class information, the developer should develop only one file for class implementation. It is easier for the developer to assign all class information in one file rather than separating the class information in two parts as declaration and definition parts and putting it in two different files. The canonical file consists of all class declaration and definition information. In a canonical file, the developer declares a class member and right after that defines it. It is obvious that developing one file for a class implementation is easier and takes less time than developing two files. It will be shown that the single file approach is more efficient in class implementation while eliminating most of the drawbacks of the two files approach.

Moreover, the single file approach has the ability of generating the header and implementation files through a program called preprocessor. The preprocessor program employs the canonical file as an input file and then produces four output files. In fact, this program separates the canonical file information into four different files: the header, implementation and two interface files. The objective of separation is to produce these four files for different users, i.e. the compiler and the client. Since the program separates these files automatically, thus the developer needs to implement only the canonical file. One of the main advantages of the single file approach is its adaptability with the current C++ compilers, which accept two files for the class implementation. Moreover, the class clients can use the interface files, i.e. the class declaration files. The interface files have all class public and protected member declarations plus comments. Therefore, the single file approach achieves information hiding as well. The preprocessor program and their outputs will be illustrated in the next two chapters.

The single file approach is also very effective for maintenance. After the

developer develops the class implementation, if the class member is changed, only the canonical file needs to be modified. The developer modifies the canonical file and then, gives the new canonical file to the preprocessor program to generate four new output files, which can be used by C++ compiler and the class clients. Therefore, comparing with the two files approach, in the single file approach it is easier for the developer to maintain the class.

3.2 Single File Approach vs. Two Files Approach

In chapter 1 it was discussed how classes are written in C++ language. In chapter 2 some problems of two files approach in C++ and other languages, which declare and define classes in two files (interface and implementation files), were studied. In this section the disadvantages of the separation of interface and implementation files in the current C++ two files approach are discussed and then, this approach is compared with the single file approach. The following class declaration and definition (Listings 3.1 and 3.2), which have been written in C++ language and have interface and implementation files, are used for this study as illustrated in the following sections.

Listing 3.1 An example of C++ class interface file (counter.h).

```
Class Counter
{
        public:
                counter(int maximum, int width);
                void    inc();
                int     carry();
                void  show();

        private:
                int maximum;
                int width;
                int counter;
                int carry_flag;

}
```

Listing 3.2 An example of C++ class implementation file (counter.cpp).

```
#include <iostream.h>
#include <iomanip.h>

Counter:: counter(int maximum, int width)
{
        // Definition of counter function
        .
        .
        .
}

Counter:: void inc()
{
        // Definition of inc function
        .
        .
        .
}

Counter:: int carry()
{
        // Definition of carry function
        .
        .
        .
}

Counter:: void show()
{
        // Definition of show function
        .
        .
        .
}
```

3.2.1 Need for Interface and Implementation Files

In the two files approach the interface and implementation files have a special format for developing. The interface file consists of member functions declaration and private data member type. The implementation file includes only member functions' definitions. The developer has to separate the class information in two different parts (declaration and definition) and to arrange two files with different formats for one class. Also, in case of modification the maintainer should modify two files. In fact, the developer and maintainer must always work with two files.

Since the private data members values and the member functions definitions must be in the implementation file, therefore the developer should also be very careful about this information to be inaccessible from the client (i.e., information hiding). Whereas, in the single file approach the developer and the maintainer put all information into one file, which is much easier to handle.

3.2.2 Duplication

Another disadvantage of the two files approach is duplication of function prototypes. Inspecting the example (Listings 3.1 and 3.2), class Counter has four member functions (counter, inc, carry, show). To declare these member functions, the functions prototype need to be written to introduce functions in the interface file. The functions prototype must also be written in the implementation file, again to define member functions. Thus, the member functions prototype (counter (int maximum, int width), void inc(), int carry(), void show()) is duplicated in two files. The formats of member functions in the header and implementation files are similar but not necessarily identical.

Consequently, each time a function prototype is changed the header and implementation files must be updated.

The single file approach avoids the duplication problem. It contains one declaration and definition for each member function of the class at the same place that includes all information of the member function, i.e. the canonical file. Since the canonical file consists of all class information it prevents repetition of declaring the information in different files.

3.2.3 Declaration and Use

In the two files approach to declare or to use a variable for the implementation, the developer must refer to two files. The interface file includes all data members' types (Listing 3.1), which are employed in the implementation file (Listing 3.2). In fact, data members' types are declared in the interface file and used in the implementation file (int maximum, int width, int counter, int carry_flag). Therefore, to employ a data member the developer should be very careful and make sure that all data members have been declared in the interface file. On the other hand, to use a data member the developer must look at its declaration in the interface file. Also, to declare a data member, the developer must look at in the implementation file to consider how the data member is declared and used in the definition part.

Moreover, the maintainer must always consider these two files. Consider the maintainer wants to change a private member's prototype, which does not affect on the interface file. Still they have to modify the interface file to change the private member's prototype. In the single file approach the developer can declare the data members exactly

before using it, and also the maintainer can easily change all necessary information in one file.

3.2.4 Interface File Contains Implementation Information

In the two files approach the interface file contains implementation information. Considering Listings 3.1 and 3.2 variables are declared in the interface file, but they are employed in the implementation file. In this example the client must know the type of *maximum* and *width* variables, which have already been declared in the counter function prototype in the interface file. This information is repeated in the data members' part. Moreover, additional variables are declared which are not necessary for the client to know. Since counter and carry_flag data members are not arguments of any function and their types and values do not have any meaning for the client therefore, it is not necessary for the interface file to have this implementation information.

Since in the two files approach the interface file is used by the compiler and the client, thus the interface file needs to have some additional information such as all data members declaration. Whereas the single file approach produces the interface file which contains all necessary data members for the client and another file named header file including all necessary information for the compiler.

3.2.5 Interface File Contains Information for Compiler

The interface file contains information for the compiler, not for the client. It includes the header of the member functions and data members' types declarations. A member function prototype has the names of the functions and arguments' types, which

are not enough for the client. In order to use a member function, the client should know what the function does. Therefore, the goal of the class and its functions should be explained earlier to the client.

In the interface file, data members' types are declared too. Each data member is an argument or is used in the implementation file for defining the function, which is not necessary for the client to know. The client needs to know only the member functions arguments' types, which are declared in the member functions' header. Thus, the interface file contains information, which is not useful for the client. One may ask then, why the interface file should be written? The interface file also contains information, which is necessary for the compiler. In the interface file, the compiler needs to have all data members and member functions, which are employed and defined, in the implementation file, to produce code for the client programs.

In the single file approach three out of the four generated files, i.e. the header, the interface, and the interface2 (or documentation) files, are for interfacing. The header file is used by the compiler and does not contain any comments. The interface files include declaration information just for the clients of a class.

3.2.6 Encapsulation

Encapsulation means that the interface file of a class does not have any representation information. In other word, clients of the class do not need to see any representation information in the interface file. Whereas, in the two files approach the interface file is developed to be used by clients and the compiler. The interface file is used for two different and conflicting purposes. The compiler needs representation

information. The interface file is written to interface the class and its functions to clients, and on the other hand the compiler uses interface file to produce code for the client programs. Since the interface file contains declarations for private, protected and public members thus, the two files approach does not provide a full encapsulation.

The single file approach avoids the above problem by generating four separate files for the compiler and clients. In the single file approach the preprocessor program is used to produce four different kinds of files, the header and implementation files for the compiler, the interfaces (documentation) files for the client.

3.2.7 Missing Information

In the two files approach information is missing from the implementation file. The interface file consists of the information regarding the access policy of each class member. The implementation file defines the class member without mentioning its access policy by the client. The developer or maintainer must refer to the interface file to see the class member access policy. Whereas in the single file approach, the developer refers to the canonical file to see the class members' access policy.

3.2.8 Comments in Header Files

The size of the header file is important for compilation time. Each header file may be read repeatedly during the compilation of a system by a large number of clients. On the other hand, a component may use some libraries. Thus, the compiler needs to compile the headers of these libraries before compiling the component. Hence, it is better to keep the size of the header file as small as possible. Also, the header file should contain

documents for clients who use this class. Documents are useful to introduce the class functions to the client. Therefore, a good documentation conflicts with reducing the compilation time.

In the single file approach, the preprocessor program produces three separate files for the compiler and the client named the header file and the interfaces (documentation) files. The programs remove the comments from the header file, whereas the interfaces files include all comments to introduce the class to the client.

3.2.9 Inline Functions

Inline functions should be visible by the compiler during the compilation of a client program, which uses this class. Therefore, the header file needs to have an inline function definition which is in conflict with encapsulation.

In the single file approach, the body of an inline function is written in the header file but it is not visible in the interfaces (documentation) files. Thus, the clients of the class never see this function definition and they do not know whether it is inline function or not.

3.2.10 Grouping Definitions

In C++ language, each class member has an attribute like private, public or protected. These attributes can be attached to a member or can make a section for each attribute and put all members with the same attribute in that section. For example, the public section has all members, which have public attribute and are visible to the client. If class members are declared in the public, private or protected sections in the header file,

they may not have the same order in the definition part (i.e., implementation file). Suppose in the definition part a public member function uses a private or protected data member thus, the public and private members should be put beside each other which is not the same order as of the declaration part which causes difficulty for the maintainer. In the single file approach, there is not any partition for the class members.

3.2.11 Clients Recompilation

In the two files approach any change in the header file leads to recompilation of all client files. Suppose B is a client of A in the two files approach, when A is modified, B must be recompiled. This way is safe but inefficient. After coding the header and implementation files of a class, most changes are related to the implementation files such as adding a private member and so on which do not have any effect on the clients files. Any change in the implementation or header file even changing a comment in the header file may cause recompilation. Hence, many client programs must be recompiled unnecessarily which takes time. It is a big problem in large systems when many clients use a system component (or a class) which has been changed.

The single file approach avoids unnecessary recompilation. It uses a cleverer criterion than a time stamp. In this approach all changes are made in the canonical file. Then, preprocessor program generates a temporary header file. The program compares the previous header file with the temporary file, which are identical in most cases and thus there is no need of recompilation. Only if there is any difference, it replaces previous header file with the temporary file, which therefore needs to be recompiled.

3.2.12 Error Handling

Error handling is one of disadvantages of the single file approach. The single file approach produces two files (header and implementation files), which are used by the compiler, from the canonical file. Therefore, the errors reported by the C++ compiler will refer to the generated files, not to the canonical file.

One way to fix this problem is to include #line directives in the generated files. This causes the C++ compiler to use the number lines for error reporting. However, not all C++ compilers support #line directives correctly.

3.3 Conclusion

The objectives of the software engineering in the coding class information are:

- providing an information hiding technique,
- simplifying the task of the maintainer,
- providing all information which the compiler needs,
- and minimizing the building time.

Unfortunately, the two files approach does not provide these objectives in a satisfactory manner, whereas the single file approach meets these objectives effectively.

Chapter 4

Usable Single File Approach for C++ Compiler

In chapter 3 a new approach was introduced which had one file instead of two files for the class declaration and definition and was named as *Single File Approach.* In this approach all class declaration and definition information is put into one file named a *Canonical* file. Current C++ compilers accept the class declaration and definition in two separate files. It will be much better to find a way by which the single file approach is used for current C++ compilers.

Each C++ class has different users such as the compiler and the client. In this thesis a preprocessor program is developed to produce four separate files for the different users (header and implementation files for the compiler and two interfaces or documentation files for clients) from a canonical file. Consequently, C++ compiler can make use of two output files of the preprocessor program, i.e. header and implementation files. As a result, the single file approach eliminates most of the disadvantages of the two files approach by providing information hiding, simplifying the task of the maintainer, providing the compiler's file with all information which it needs, minimizing building

time, etc. This chapter describes what the preprocessor program does and what it input and output files look like.

4.1 Single File Approach Implementation for C++

A program named as preprocessor is developed in this thesis. The goal of the preprocessor program is to employ the single file approach for the C++ compiler. The C++ compiler does not accept a single file for the class declaration and definition, rather they need two separate files. One file contains class declaration and the other file consists of the class definition information.

The preprocessor program has two components named as *Parser* and *File Generator*. The parser gets the canonical file as an input file and parses the input file using a top-down recursive-descent parser, then it makes a tree from the input file which has a class declaration and definition information. The tree has several branches, and each branch contains a specific part of the class information such as library, class name, class members, etc.

For instance, each class can be a client of other classes; in this case it should introduce the name of other classes header files at the beginning of its information with "library" or "supplier" keyword. When the parser reads the first "library" keyword, it makes a new branch as library and adds the header files name as subbranches to this branch. Consequently, if the parser reads more "library" keywords, new subbranch will be added to the library branch to assign the new header file name to the tree.

After the parser completes the tree, the file generator will use this tree to generate four different files. The file generator goes through the tree's branches and decides which

information goes into which file. The preprocessor program should be written to get the canonical file as an input file, but first the input file grammar must be defined. The input file grammar should declare and define the class perfectly. Thus, it is necessary to know what the canonical file needs for the class declaration and definition.

4.2 Canonical File Grammar

C++ language can be used to implement the canonical file presented in this thesis. A context-free grammar expressed in the Extended Bacus-Naur Form (EBNF) is used for canonical file grammar. The complete canonical file grammar has been written as Listing 4.1 which shows the canonical file syntax. A class information should be declared and defined so that it matches with this syntax.

Each class may use some libraries or header files in the beginning, thus the canonical file can have a statement, which is defined in the input grammar as following:

{"library" Identifier{, Identifier}}

{"supplier" Identifier{, Identifier}}

The statement between "{" and "}" signs may occur zero or more times. Then, the above statements mean a class may employ zero or more libraries. The "library" and "supplier" statements are defined exactly the same as the "#include" statement in C++ language. The "library" statement defines all libraries file and the "supplier" statement introduces all header files, which are used in the canonical file. Everything between " " signs must appear as a keyword which is allowed in the language.

Listing 4.1 : The canonical file grammar

```
class -> { "library" Identifier{, Identifier}";" }  [//comment]
        {//comment}
        { "supplier" Identifier{, Identifier}";" }  [//comment]
        {//comment}
        { "const" Type Identifier"="Identifier";" }  [//comment]
        {//comment}
        { "enum" Identifier "{" Identifier { "," Identifier } "};" }  [//comment]
        {//comment}
        ("class" Identifier [":" ["private" | "public"] Identifier] ";")
        { "inherit" ( "public" | "private" ) Identifier }  [//comment]
        {//comment}
        { "friend" "class" Identifier }  [//comment]
        {//comment}
        { Feature}

Feature→[ "public" | "private" | "protected"]
        [ Type [Type] ] ["constructor" | "destructor"]
        [ "*" | "&"] Identifier [ [ "*" | "&" ] Identifier ]
        ( ";" | "["Expression"]" ";" | "("[ParameterList]")" {//comment} "{"Body "}" {//comment})

ParameterList →Type ["*"|"&"]Identifier | {"," Type ["*"|"&"]Identifier }
Type → Identifier
Expression → Identifier | Integer
Identifier → {Char}
Char → A |B |C |...|Z |a |b |c |...|z |Integer |Idsign
Integer → 0 |1 |2 |3 | ... |9
Sign → + | - | * | / | ^
Idsign → _ | # | .....?
Body → Identifier | Sign | ";" | {//comment} | "{" [Body] "}"

{ ... } Zero or more times
[ ... ] Zero or one time
| Alternate
// Operation Comment
```

Thus, to introduce the libraries or header files, which a class uses, the "library" or "supplier" keyword should appear in the beginning of the statement. The Identifier indicates a word consisting of letter, or digit, or some special characters, which can be used in the file name such as underscore, dollar sign, etc. Identifier introduces one or more library or header file name.

Similar to C++ language, in the present grammar if two slash signs "//" appear together. there will be a comment. Everything after these two slashes up to the end of the line will be considered as a comment. The comment can be written in the beginning of a line or at the end of a statement, i.e. after semicolon. For functions the comment is written between the function name and function body. The function body can have comments too. In the canonical file grammar comments are shown by the following statements:

{//comment}

[//comment]

Which {//comment} means zero or more lines of comment and [//comment] means zero or one comment.

Since the canonical file is written for a class information, thus the class name should be given in this file. Like in C++ language, first the "class" keyword and then the class name should be introduced. In the grammar used in this thesis (Listing 4.1) class name statement is such as:

("class" Identifier ";")

which introduces the class name. The class keyword should be followed by an Identifier which is the class name.

Each class may use some constant identifiers. The canonical file may have some constant identifiers as well. In the canonical file grammar the class constant identifiers are defined after declaration of the class name as follows:

{ "const" Type Identifier "=" Identifier; }

It means each class may have some constant identifiers and each identifier is defined in one line, which has "const" keyword. The type of the variable and the Identifier followed by = sign is next. Then, another Identifier and at the end the semicolon sign are written. The first Identifier is the name of the constant identifier and the second one is a number, or character string, or the name of another constant identifier.

If the class has some enumerated variables, it can be defined as:

{"enum" Identifier "{" Identifier {"," Identifier } "};"}

which introduces an enumerated type variable, exactly in the same way as of C++ language. It has "enum" keyword, which is followed by an Identifier as enumerate variable name then "{" sign. After that, it has some Identifiers to introduce all kinds of enumerate variables. The first brace ("{") is a symbol that appears in the C++ program. The second brace ({) is a metasymbol that means there can be zero or more enumerated variables. Each kind of enumerated variable (Identifiers) is separated by a comma sign (",") and at the end there is a brace and a semicolon sign ("};") to show the end of this enumeration. Each class can have some enumeration variables and each of them should be declared in the same way as mentioned above.

Each class can be a derived class, which should be indicated in the canonical file. Keyword "inherit" can be used to introduce the derived class and its base class. The base class can be public or private. To show the access policy of the base class, after the

"inherit" keyword and before the name of the base class, the keyword "public" or "private" must be used. Thus, after the keyword "inherit" we will have the keyword "public" or "private" which is shown in the grammar as ("public" | "private" | "protected"). Sometimes the access policy is not mentioned explicitly thus, the default will be private. Then, there will be an Identifier, which is the base class name. Finally, there may be a comment. The inheritance line of the grammar used here is as:

{"inherit" ("public" | "private" | "protected") Identifier}

Each class may be a friend (in the C++ sense) of some other classes. To show a class is a friend of another class, after introducing the class name and its inheritance, its friend class's name should be mentioned. The grammar introduced here does this job in the same way as of the C++ language. To introduce the class friend's name, there should be two "friend" and "class" keywords, which are followed by an Identifier, i.e. the name of another class. In the canonical file syntax this is done as:

{ "friend" "class" Identifier }

The braces emphasis that each class may have more than one friend.

Each class has some members. A class member is a data member or a member function of the class. After introducing the class name and inherit part, the members of the class are introduced. This part of the canonical file is named as a feature part. The canonical file may have {feature}, which means each class member is declared and defined with a feature structure. Each class may have zero or more members, therefore there may be zero or more feature structures at the end of the canonical file.

Now, the feature must be defined. The feature may begin with the keyword "public", "private", or "protected" which can be shown as following:

["public" | "private" | "protected"]

Default is "private", which means if the feature does not begin with " private", "protected", or "public" keyword then, the access policy of this member will be private. The access policy keyword follows with Type, which is the type of the member. In the canonical file grammar the Type can be:

Type → int | char | float | double | short | long | void | const | ifstream | ofstream | virtual

There may be two different kinds of Types together like: "virtual void" or "long double". The statement [Type [Type]] used in Listing 4.1 means the Type may appear once or twice for each class member. After Type, there is one or two Identifiers, and each of them can be a pointer or a refrence, thus:

[* | &] Identifier [[* | &] Identifier]

This means there is at least one Identifier which may be a pointer or a reference and may have another Identifier which can be a pointer or reference too. If the first Identifier is a pointer or reference there can not be any more Identifier after that. If the Identifier is a simple data member there will be a semicolon after it which indicates the end of data member's declaration. There will be a "[" Expression "]" when the Identifier is an array. Expression indicates the size of the array which may be a number or an Identifier. In the grammar, the Expression is declared as following:

["public" | "private" | "protected"] Type Identifier"["Expression"]";

Expression → Identifier | Integer

If the member, which has already been declared, is a function, two parentheses "(" and ")" must be written. There may also be some parameters between these two parentheses. In the canonical file grammar the function parameters are introduced as "("

[ParameterList] ")", which means that the function may have some parameters. The ParameterList is one or more series such as:

Type [* | &] Identifier

Consequently, the ParameterList is introduced as:

ParameterList→Type [* | &] Identifier | {,Type [* | &] Identifier }

After ")" sign, the Body of the function should be written. In fact, Body is the definition part of a member function which shows exactly the way that the function does the specific task. In the canonical file grammar the Body of a function is shown as following:

["public" | "private"] [Type [Type]] Identifier([Expression]) {Body}

Body → {Identifier | Sign | ; | { [Body] }

The feature may be class constructor or destructor function. In this grammar the constructor and destructor are defined as:

"public" constructor Identifier(ParameterList);

"public" destructor Identifier();

The destructor can not have any parameter.

4.3 Parser and File Generator

The canonical file is as the parser input file. If the canonical file is as Listing 4.2, the parser reads the canonical file and produces a tree as Figure 4.1. Then, the file generator employs the tree and generates four different output files. The preprocessor program has three steps:

a) Parsing the Input,

Listing 4.2 A sample of a canonical file.

```
library stdio, stdlib;

class Test : public T1
        // A class for testing things
{
        private int Wcount;
                // Counter for widgets
        public void f1()
                // A pointless function
        {
                if (flag)
                {
                        return 0;
                }
                // { test braces }}
                else
                {
                        return 1;
                }
                char bb == " {{ test braces } ";
        }
        private double Wname;
        private char *strtest;
        protected char prot;
        private float g (int n, float x, char *ts, char arry[arrylen], char &amp)
                // Another pointless function
        {
                x = sqrt(n);
        }
}
```

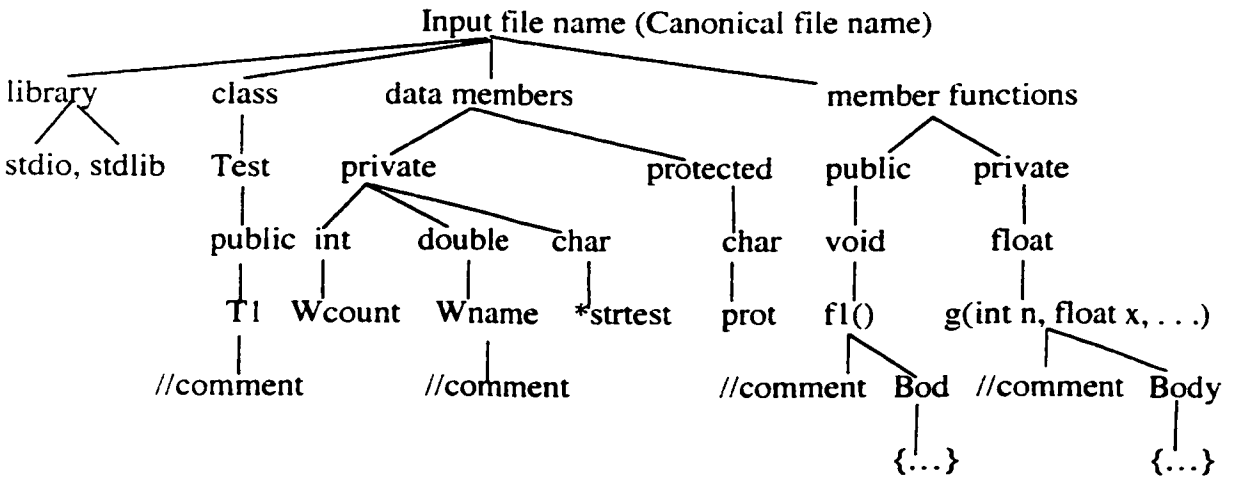

Figure 4.1: The output of the parser, i.e. a tree.

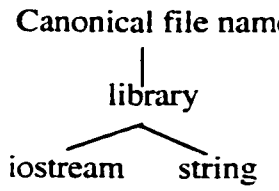

Figure 4.2 The library subbranch.

b) Producing Abstract Syntax Tree, and

c) Writing four output files (header, implementation and two interfaces or documentation files).

The parser parses the input file from top to the bottom. The parser is top-down and uses recursive-descent. The parser reads the canonical file character by character and checks the content of the input file with the syntax as shown in Listing 4.1. If the input file does not match with the syntax, the parser will send a message regarding the error. Otherwise the parser produces a tree corresponding to the input file. For example, if the input file has library statements, as

library iostream,string;

the parser makes a tree with a branch named library (Figure 4.2) and assigns all library files names as the library subbranches.

As another example if the input file has some member functions, the parser makes a branch named as member function and subbranches for each function of the input file (Figure 4.1). The size of the parse tree is limited only by the size of the memory.

Using the tree produced by the parser (Figure 4.1), the file generator generates four different output files. A file for the compiler as implementation file that has all class member functions' definition information. It also generates a header file for C++

compiler which has all useful information for compiler to compile class clients file such as all data members as well as member functions prototype without any comment. The third file is documentation (interface) file, which contains all class public members declaration information in addition to all corresponding comments to this information. The fourth file is the same as documentation file plus protected members declaration. This file will be used by the client who wants to use the class as a base class. These files are listed in Listing 4.3 – 4.6. The difference between Figure 4.5 and Figure 4.6 is the protected member part.

Listing 4.3 Implementation file: output of the file generator.

```
#include <stdio.h>

void Test :: f1()
{
        if (flag)
        {
                return 0;
        }

        else
        {
                return 1;
        }
        char bb == " {{ test braces } ";

}
float g (int n, float x, char *ts, char arry[arrylen], char &amp)

{
        x = sqrt(n);

}
```

Listing 4.4 Header file: output of the file generator.

```
#include <stdio.h>

class Test : public T1
{
        public:
                void f1();
        private:
                int Wcount;
                double Wname;
                char *strtest;
                float g (int n, float x, char *ts, char arry[arrylen], char &amp)
        protected:
                char prot;
}
```

Listing 4.5 Documentation(interface) file: output of the file generator.

```
#include <stdio.h>
class Test : public T1
        // A class for testing things
{
        public:
                void f1();
                // A pointless function

}
```

Listing 4.6 Documentation2(interface2) file: output of the file generator.

```
#include <stdio.h>
class Test : public T1
        // A class for testing things
{
        public:
                void f1();
                // A pointless function
        protected:
                char prot;

}
```

Chapter 5

Implementation of Parser and file Generator

To employ the single file approach by the C++ compiler the canonical file must be split up into four files as interfaces. header, and implementation files. In this thesis, a parser is developed to parse the canonical file and then a file generator separates declaration and definition information and generates four different files. The parser and the file generator are a set of programs, which are written in the C++ language. In this chapter it is shown how the parser and file generator are implemented and how the parser reads the canonical file and the file generator produces the output files (Appendix A).

5.1 Input and Outputs Files

As illustrated in chapter 4, the input file of the parser is the canonical file. The canonical file includes all class declaration and definition information. A sample of a canonical file is shown in Listing 4.2. It is supposed that canonical file consists of only the information of one class.

The output of the parser is a tree. The parser parses the canonical file and produces a tree as shown in Figure 4.2. This tree assigns a branch for each section (i.e. library line, data members, member functions, etc) of the input class and then assigns all information of this section as subbranches of this branch which named as the section of the class. A sample of a branch and its subbranches is shown in Figure 4.1.

The input of the file generator is a tree produced by the parser. The file generator traces the tree and divides the information into four files. The first file, i.e. implementation file, includes all member functions definition. The compiler needs private, protected, and public member declaration plus all member functions prototype information without any comment. Thus, the file generator generates the second file (header file), which consists of this information. Documentation file with extension "doc" (the third file) consists of all public class member declaration information plus comments. The client of the class will use the documentation file. The fourth file includes all public and protected member information and comments which will be used by class client who wants to use the class as a base class. This file is saved as the second interface file with extension "int".

5.2 Preprocessor Program Classes

The preprocessor program is written in C++ language (Appendix A). Also in this program several classes with C++ class implementation approach are developed. The class declaration and definition are split up into two separate files named as the interface (or header) file and implementation file. The classes name and their functions are explained in this section. All classes in the program have four functions, i.e. gen-interface, gen-interface2, gen-header, and gen-impl, which produce different information for the four different output files.

The preprocessor program has a class named Token. The class Token holds the current state of the scanner and is also responsible to read tokens from the canonical file. A token can be anything (i.e., a keyword, or an identifier, or a sign, etc) except blank. The class Token has an enumerated type named Kind which consists of all kinds of a token. This class has data members as buffers and member functions. The class Token includes some functions too such as set_file(), newline(), skip(), read(), read_body(), error() and match(). Token class has some buffers for keywords, identifiers, the input file line, and the text of a function body and a buffer for the text of comments. The buffers sizes are defined by 120 characters.

The canonical file (input file) is opened and read by the set-file() and read() functions of the class Token. The class Token employs the newline() function to get one line of the canonical file and register this line into a buffer. Then, it uses the read() function to read the buffer, character by character to find a token. The blanks of the input file line are skipped by the skip() function of the class Token. The error() function is another function of the class Token which checks whether the input token is acceptable or

not. After reading the input token, the read() function will search in the token kind list to find the input token kind. If the read() function does not find any token kind that matches with the input token, it will send an error message. The class Token is implemented in two files named token.h and token.cpp. Token.h file contains the class Token's declaration information and Token.cpp file includes the class Token's definition information.

Another class of the program is the class List. This class has five member functions and two data members. The class List has a member function named List(). List() function constructs an empty list, which has pointers to both ends of the list so that new nodes can be added at the end of the list. Another member function of the class List is append(). Append() function appends a node to the end of the list. This function gets the pointer which points to a node and adds this node to the end of a list.

Also, the class List has four more member functions to generate four different information of a list for four files, i.e. gen-interface(), gen-interface2(), gen-header(), and gen-impl() functions. The gen-interface() function selects some of the list information which is useful for the client and put them into the interface output file. The gen-interface2() function selects some of the list information which is useful for the client how want to use the class as a base class and put them into the second interface output file. The gen-header() function chooses some of the list information which is needed by the compiler and put them into the header output file. Finally, the gen-impl() function selects some of the list information which is related to the input class member function definition and put them into the implementation output file. The class List has two pointer data members to point the first and last nodes of the list.

The class Listnode is another class of the preprocessor program. This class is a friend of the class List. The Listnode() function is a member function of the class Listnode. The Listnode() function gets a node as a parameter and adds a new node to a list with a pointer to the given node.

In the preprocessor program the class List and the class Listnode are implemented in two files named list.h and list.cpp. List.h file consists of all declaration information of the class List and the class Listnode. All definition information of the class List and the class Listnode are implemented in the list.cpp file.

The program has also other interface and implementation files named as comment.h, comment.cpp, feature.h, feature.cpp, filename.h, filename.cpp, library.h, library.cpp, module.h, module.cpp, node.h, node.cpp, parameter.h, parameter.cpp, supplier.h, and supplier.cpp. In these files, several classes are implemented and each of them generates a special kind of node for a tree.

Class Node is an abstract base class for all nodes of the tree shown in Figure 4.1. All derived classes of the class Node must implement gen-interface(), gen-interface2(), gen-header() and gen-impl() functions. The file generator employs these functions to generate texts for the two interface (or documentation) files, the header file, and the implementation file respectively. The class Node has a protected data member to point to a token.

The class Module is a derived class of the class Node. The parser employs this class to generate the root node of a tree. In fact the class Module is a module corresponding to a single class. The class Module includes a member function named as Module(). The Module() function creates a node as the root of a tree. It employs the class

Token to read the input file's tokens.

First, the class Module looks for the comment. If current token kind is COMMENT it makes a list for the comment and calls class Comment to append a node to the list. Then, the class Module adds all other comments, if there is any, to the comment list. The class Comment takes current token, which is a comment, as input and saves it in a data member and puts it into the two interface files. The class Module checks if there is any comment at the end of each statement. Thus, the developer can put as much as comments at the end of every statement.

If there is any library keyword, the class Module makes a new list, which consists of all libraries file name and then employs class Library to assign a branch for "library" directives. The class Module points to the library list as a branch of the tree.

If there is any supplier keyword, the class Module makes a new list which consists of all suppliers file name and then employs class Supplier to assign a branch for "supplier" directives. The class Module points to the supplier list as a branch of the tree.

The class Module has a data member, which is a character string, to register the class name. The class name is a branch of the tree. Also, the class Module puts comments into the comment list, which are branches of the tree.

The class Module employs the class List to create a new list named features list. The features list includes all class data members and member functions information. Each node of the feature list is created by the class Feature. The class Module points to the features list as a branch of the tree.

The class Module has four member functions, i.e. gen-interface(), gen-interface2(), gen-header(), and gen-impl() functions. These four functions generate four

different texts for four different output files. The gen-interface() function puts the class name and comments into the output interface file. This function also employs the class Feature's gen-interface() function to generate public data members and member functions prototype information plus comments. The gen-interface2() function is exactly as gen-interface() function. In addition it also produces all protected members.

The gen-header() function of the class Module puts #ifndef and #define statements, which are followed by the canonical file name, into the output header file. Then, it writes the class name in the output header file. It also employs the class Feature's gen-header() function to produce all class data member and member functions prototype needed by the compiler. Then it prints them in the output header file. At the end, the gen-header() function writes the #endif statement.

The class Feature is a node for a feature that can be a data member or a member function. The feature includes the "private", "public", or "protected" keyword and an Identifier as the data member name. Also for a member function, the feature includes "private", "public", or "protected" keyword followed by an Identifier as the member function's name and two parenthesis "( )" which may include some parameters. The feature may include the body of the member function and the comment as well. It can also be either a class constructor or a class destructor.

The class Feature has a member function named Feature() which parses the input file to find a data member or a member function. This function also employs the class List to put the member function's parameters into a list named as Params list. The Feature class then uses the class Parameter to create a node for Params list.

The class Parameter is a node for a single parameter. It is also a derived class of

the class Node and has a member function named Parameter() function. The Parameter() function creates a node to put the input class member function's parameter into it. Each function parameter is a type followed by an Identifier and may have a comma at the end. The class Parameter has four more functions to generate the text for four output files.

5.3 Parser Algorithm

The parser is based on a top-down recursive-descent parser. First, the parser asks the user to enter the input file name. The parser makes an object for the class Token. Then, the parser calls the set-file() function of the class Token to open the input file. The set-file() function opens the input file and then employs the newline() function of the class Token to get an input file line and put it into a buffer. The buffer is an array with a specific length. In the parser, it has been assumed that each input file includes only one class information.

The parser creates a new object of the class Module by calling the Module() function of the class Module. This object is a node such as a root node of a tree and points to the input class information as the tree's subbranches. Consequently, the root node of a tree, which is the output of the parser, is created. The Module() function makes a pointer for the class Token object. Then it calls the class Token read() function to read the buffer containing the input file's line to find a token. A token can be an identifier, keyword, sign, punctuation, or so on.

After the program finds a token, it looks for its kind. The parser checks for a comment at the end of each statement and one or more comment lines between two statements. Whenever the program parses two slashes "//" then, it takes everything after

these signs as a comment up to the end of the line. It makes a list whose node consists of a comment.

If the token is "library" or "supplier" keywords, the program creates the new lists named libraries or suppliers for inclusion of all "library" or "supplier" statements. Also, if the input file has more than one "library" or "supplier" line, the program adds these statements in the corresponding list. The class Module has a pointer to both libraries and suppliers lists.

After parsing all "library" and "supplier" statements, the program looks for "class" keyword, which is followed by an identifier as class's name. When the program finds the "class" keyword it looks for a token (an identifier) and puts the Identifier into a data member of the class Module, which is a character variable. The class may be a derived class. In this case after the class name there is inheritance part which can be an access policy to the base class plus the name of the base class. If the class is a derived class, the parser registers the inheritance access policy and base class name in inheritance and inhrname variables. After the parser puts the class name and/or base class name into the class Module variables, it may parse a comment. When the program parses a comment, it puts the comment into a list named as "comments". At this step, if the current tokens, which are read, are "friend" followed by "class" keyword and an identifier, the program accepts it as a friend class.

Now, the program should read the features. The feature is a data member or a member function. In the canonical file, a developer can assign individual "public", "private", or "protected" keyword for each member, which is an easy way of writing the class members information. Hence, each feature in the canonical file begins with one of

these keywords: "public", "private", or "protected". The program makes a list of features, and appends new members to the feature list while it parses "public", "private" or "protected" keywords followed by a feature.

The program reads the input, token by token. After it reads "public", "private" or "protected" keyword, it expects to have the type of the feature otherwise the program sends an error message. In the case that it parses a type name such as int, char, void, etc. the program should read the name of the feature as an identifier and put it into a variable. If after the type of the feature the program does not parse an Identifier, it sends an error message.

After the program reads the feature name, it may face a left parenthesis, which means that the feature is a member function. If the program parses a semicolon or an array, which includes a left bracket, Identifier, and a right bracket, the feature is a data member. In the first case, i.e. the feature is a member function, the parser should read a left parenthesis and a list of parameters or right parenthesis. After the program parses a member function feature, it makes a new list (params) to put all function's parameters into it. The parser uses the class Parameter to generate a node for each parameter and appends this node to the params list. The program has a *while loop* to read all function's parameters till it faces a right parenthesis. Then, it may parse a comment and consequently puts the comment into the comment list.

After the program reads the function prototype, it should parse the body of the function or semicolon. When the program parses the left bracket, it is actually parsing the body of the function. The parser dose not check syntax of a function body; it just copies it. An important point in copying the body of a function is the number of brackets. Each

function's body may have several left and right brackets. The program, first parses the first left bracket then counts the number of brackets which has read, and whenever it parses a right bracket it deducts one from the number of brackets. Also, when the program parses the first bracket of the function's body it puts everything, which has read, into a string named as the body. When the program reads a right bracket and then the number of brackets becomes zero, the body of the function is finished. The program goes back to search whether there is another feature or that is the end of input file.

5.4 File Generator Algorithm

The file generator generates four files by using the abstract syntax tree produced by parser. It creates four output files with the same name as of the input file of the parser but with different extensions. For instance, if the input file name is test, consequently the program creates four new files as test.imp, test.hdr, test.int, and test.doc. The test.imp and test.hdr are class implementation and header file respectively. The test.doc and test.int are class interface files for two different clients of the class.

The file generator uses the class Module gen-interface() function to generate an interface file which explains what the class provides to the clients of the class. The interface file consists of all class public member declaration in addition to comments. The gen-interface function prints the libraries and suppliers lists into this file. These two lists contain all canonical file's library and supplier file names. Then, it prints the "class" keyword and the name of the class stored in the class Module classname variable. Consequently, the program puts the comments into the interface file if there is any comment after the class name line or public member.

Finally, the class Module gen-interface() function prints the features list by using the gen-interface() function of the class Feature. The class Feature gen-interface() function selects public data members plus public member functions prototype and comments of these members from the features list. In the input file, the developer does not need to separate public and private members and put them into separate parts. In fact, the file generator separates public, protected and private members and puts them into different sections of the output files (interfaces and header files).

The second output file is the header file. The file generator creates the header file for the compiler. The header file is like C++ header file without comments. The file generator employs the class Module gen-header() function to produce the output header file. The class Module gen-header() function uses the libraries and suppliers lists to print all library and suppliers file name. It prints each library or supplier file name in one line which is started with "#include" keyword. Then, it writes the class name, and at the end using the class Feature gen-header() function to print the features list for the output header file. The class Feature gen-header() function selects member functions prototype plus all data members declaration and inline functions of the input class from the features list and puts them all together into the header files.

The third output file is the implementation file, which contains libraries and suppliers file name written with "#include" keyword and all class functions definitions without comments. This output file is used by the compiler and is hidden from the class clients. The file generator employs the class Module gen-impl() function to produce the output implementation file. The class Module gen-impl() function uses the class Feature gen-impl() function to select the input class functions prototype and definition (body)

from the features list and prints them into the output implementation file. The class Feature gen-impl() function prints the function type, the input class name, two colon signs ("::"), and the member function's name followed by the left parenthesis into the output implementation file. Then, it employs the class Parameter gen-impl() function to print the input class member function's parameters. After printing the member function's parameters, the class Feature gen-impl() function prints a right parenthesis and at the end, it prints the body of the input class member function.

The fourth file is interface2, which is exactly like interface (documentation) file in addition to protected members used by the class client.

After generating the four output files, the file generator closes all files. This is the end of preprocessor program task.

Chapter 6

Discussion

In this chapter, we discuss whether the single file approach is a useful product for the developer to use. As illustrated in chapter 3, the two files approach has many drawbacks, whereas the single file approach does not have these disadvantages. Since the single file approach needs only one file for C++ class declaration and definition, it is easier for the developer and maintainer to write and handle just one file. They put all class information into one file and do not need to separate the class declaration and definition parts neither does they need to make two different files for each part. As described in chapter 5, in the single file approach, the developer needs to write only the canonical file and implement all class information into this file. Thus, the developer can develop a class information in a shorter time and with an easier way.

The new approach avoids the need to repeat some information such as a member function prototype. In the single file approach each member function is declared and then is defined right after the declaration only in the canonical file. Thus, it does not need to declare a member function prototype anywhere else whereas, in the two files approach a

member function prototype must be declared in two different files, i.e. the interface file and the implementation file.

In the single file approach a data member is declared and then is used in the canonical file. As it is shown in chapter 4 and 5, when the developer needs to use a data member in the definition of a member function in the canonical file, he/she should declare the data member first and right after that employs it. Hence, in the single file approach the developer declares and uses a data member in the canonical file. Thus, it does need to refer to two different files. In the two files approach, a data member is declared in the interface file and is used for the definition of a member function in the implementation file. Therefore, the developer needs to refer to two files to declare and use a data member.

In the two files approach, the interface file contains information for the compiler moreover, the interface file consists of information for the class client. These two kinds of information conflict with each other. The interface file is used by the compiler and must have the information for the compiler. The compiler needs all class member declarations such as private data member which are not accessed by the client. Also, the interface file should contain the information for the client to introduce the class to them such as comments, which are necessary for the client. Thus, the compiler should read the comments wasting its reading time. The client sees the information, which is not necessary for him/her such as data member declaration. On the contrary as illustrated in chapter 4 and 5, in the single file approach the preprocessor program produces four different files for the compiler and the client, i.e. the header file and the documentation (interfaces) files. The header file contains all class member declaration, which is

necessary for the compiler, without any comment. Thus, the compiler does not need to scan the comments and actually it saves the compilation time. The documentation file contains all class public and protected member declaration plus all comments, which are useful for the client. Also, the class clients can not see any private data member declaration and hence, the single file approach provides a full encapsulation.

In the two files approach the implementation file does not contain any access policy for the member functions. The developer and maintainer should see the interface file to find out what the access policy of a member function is, whose definition part is in the implementation file. In the single file approach the access policies, declaration and definition information of all members functions are mentioned in the implementation file. Thus, the developer and maintainer do not need to return to two files to see the access policy and the definition of a member function.

In the single file approach, the file generator puts the inline functions into the header file, which is used by the compiler. Also, the inline functions must be hidden from the client, hence the interface (documentation) file does not have any inline functions information. In the two files approach, inline functions are visible to the client because the interface file should contain this information which is used by the compiler.

In the two files approach, when the class maintainer changes any item of the class, even if it does not have any effect on the client program, the client program must be recompiled. This is a big problem in a large system, because if a class is used by many clients, after a small modification in the class all clients should be recompile. In the single file approach, as explained in chapter 5, the maintainer only changes the canonical file and then the preprocessor program produces interfaces, header and implementation

files. The client program must be recompiled only if the header file is changed, otherwise the client program does not need to be recompiled.

In chapter 1, it was mentioned that there are some advantages in the two files approach such as separation of the interface and implementation files. The single file approach keeps these advantages of the two files approach while eliminating its drawbacks. In the single file approach the class declaration and definition are exactly the same as those in the two files approach. Thus, the developer does not need to learn any syntax to write the class information in the canonical file of the single file approach.

The single file approach is a good approach for the developer. The developer can save the class developing time by an easy way when the single file approach is employed. As illustrated in chapter 4 and 5, the developer writes just one file (canonical file) for the class declaration and definition, thus all class information must be put into one file. The developer declares a class member whenever it must be used or defined. The most difficult part to develop a class in the two files approach is to separate the interface and implementation files, whereas in the single file approach the separation is done by the preprocessor program automatically. The separation of the interface and implementation files in the two files approach has many problems regarding encapsulation and information hiding. The two files approach does not achieve complete encapsulation and information hiding. Whereas in the single file approach the four output files satisfy complete information hiding and encapsulation.

It is strongly advised to the C++ class developers to use the single file approach, which is a good product for class implementation. The single file approach has similar syntax to the two files approach and is very easy for the developer and maintainer, who

are familiar with the current C++ class implementation. Moreover, this new approach performs class information separation part automatically by employing the preprocessor program. Thus, in the single file approach the developer declares and defines all class members without any attention to the separation, information hiding, encapsulation, etc. Since the developers need to take care of only the class member declaration and definition therefore, it is efficient for them to use this new product. The single file approach is an easy way for developing the C++ classes with an efficient way.

Chapter 7

Conclusion

C++ class implementation was studied in this thesis. The main goal of this thesis was to introduce a simple and efficient class implementation approach, which could be easily implemented and maintained. These objectives have been successfully achieved through this research, the accomplishment of which is briefly summarized in the following.

A single file approach was developed for the C++ class implementation to eliminate the current C++ class implementation drawbacks and to keep its advantages. By using the single file approach the developer creates only one file for class information instead of two files in the current C++ class implementation (two files approach). Since the single file approach needs only one file for a class information then, it reduces the class development and modification time.

Since the single file approach is similar to the two files approach, the developer who is familiar with the two files approach can also easily use the single file approach. In the single file approach the developer declares and defines class information into one file

named as the canonical file. The syntax of class declaration and definition in the canonical file is the same as class declaration and definition syntax in the interface and implementation files of the two files approach.

The new approach achieves full information hiding. By separation of the two files approach's interface file into two files for compiler and client, i.e. header file and documentation (interface) file, the single file approach prevents the client to see some private class members such as private data members. In the documentation file all class description are available to clients by using comments. These comments illustrate the class task. On the other hand, the header file is used by the compiler can have some private information such as private data members to declare all class members and it does not include any comment which is not necessary for the compiler.

In order to keep the two files approach advantages such as separation of the class information into several files, the single file approach employed preprocessor program to separate the class declaration and definition parts from the canonical file. It separates the class information into four different files instead of two files to achieve encapsulation and information hiding. In the separation of the class information, the single file approach overcame the drawbacks of the two files approach. For instance, in the two files approach the implementation file does not have any access policy of member functions, but in the single file approach the canonical file contains them.

Employing the preprocessor program, the single file approach reduces the developer's task. In the two files approach the developer should separate class declaration and definition parts into two different files which takes time for developing the class. The developer must be very careful to write these two files and put class declaration into the

interface file and class definition information into the implementation file. In the single file approach the separation of the class information is done automatically by using the preprocessor program, and the developer provides all class information to only one file (canonical file) and then the preprocessor program separate the declaration and definition parts efficiently.

As a result, the single file approach is a similar to the two files approach but without its drawbacks. Since the single file approach does the separation of the class declaration and definition parts automatically then, the developer needs to spend less time to develop the class information. Hence, the single file approach is an easy, efficient way to implement the class information in C++ programming language.

7.1 Suggestions for Future Extension of this Research

The following studies will be appropriate for the continuation of this research.

- Developing a new program to generate the canonical file automatically. Since the syntax of the canonical file is known, thus a program can be written to ask the developer to enter the class information in order.
- Studying the single file approach for other object oriented languages, which are using the two files approach such as Ada, Eiffel, Smalltalk, Modula-2. The current single file approach is developed for C++ programming language. Application of the methodology proposed here for other object oriented languages can be investigated in future.

References

[1] Anuff, Ed, 1996, *Java Source Book*, John Wiely & Sons, Inc., p.p. 119-137.

[2] Arnold, Ken and Gosling, James, 1998, *The Java TM Programming Language*, Second Edition, Addison Wesley.

[3] Daconta, Michael C., 1996, *Java for C/C++ programmers*, John Wiely & Sons, Inc., p.p. 17.

[4] Ellis, Margaret A., and Stroustrup, Bjarne, 1990, *The annotated C++ reference manual*, Addison-Wesley.

[5] Flanagan, David, 1997, *Java In A Nutshell*, Second Edition, O'Reilly & Associates, Inc., p.p. 18-20, 71, 72.

[6] Lalonde, Wilf R. and Pugh, John R., 1990, *Inside Smalltalk*, Volume 1, Prentice Hall, p.p. 19.

[7] Meyer, Bertrand, 1997, *Object-oriented Software Construction*, Second Edition, Prentice Hall PTR., p.p. 24,25.

[8] Meyer, Bertrand, 1992, *Eiffel the Language*, Prentice Hall International (UK) Ltd.,.

[9] Meyer, Bertrand, 1988, *Object-oriented Software Construction*, Prentice Hall International (UK) Ltd., p.p. 22

[10] Parnas, David L., December 1972, *On the criteria to be used in decomposing systems into modules*, Comm. ACM, 15(12) : 1053-1058, p.p. 4.

[11] Schildt, Herbert, 1987, *Advanced Modula-2*, Osborne McGraw-Hill, p.p. 14, 37, 38.

[12] Sommerville, Ian, 1995, *Software Engineering*, 5th Edition, Addison-Wesley.

[13] Thomas, Pete and Weedon, Ray, 1995, Object-Oriented Programming in Eiffel, Addison-Wesley, p.p. 41,42.

[14] Volper, Dennis and Katz, Martin D., 1990, *Introduction to programming using Ada*, Prentice Hall, p.p. 257-274.

[15] Watt, David A. And Wichmann, Brian A. And Findlay, William, 1987, *Ada language and methodology*, Prentice-Hall, p.p. 151-160.

[16] Wilson , Leslie B. and Clark, Robert G., 1993 *Comparative programming languages*, Second Edition, Addison-Wesley, p.p. 2,12,17.

Appendix A

Preprocessor Program, Header Files, The Input File (Canonical File), and Output Files

****** preproc.cpp program ******

```
#include <iostream.h>
#include <string.h>
#include "node.h"
#include "module.h"

// This is the main program for the preprocessor.

// If 'trace' is non-zero, the program will display a message
// each time a new node is added to the tree.
int trace = 1;
bool newhead = false;char buffer[BUFLEN];              // input file
line bufferchar bufhdr[BUFLEN];char buftemp[BUFLEN];

// The 'token' holds the current state of the scanner and is
// also responsible for reading tokens from the input.
Token token;

void main ()
{
      // Buffers for file names
      char filename[FILENAMELEN];
      char infilename[FILENAMELEN];
      char docfilename[FILENAMELEN];
      char tempfilename[FILENAMELEN];
      char hdrfilename[FILENAMELEN];
      char impfilename[FILENAMELEN];
      char intfilename[FILENAMELEN];

      // Ask user for the input file name.
      // Currently the only input file is called 'test'.
      cout << "Enter name of input file without extension: ";
      cin >> filename;

      // Create names for input file and output files.
      // The header file has extension '.hdr' (rather than '.h')
      // and the implementation file has extension '.imp'
      // (rather than '.cpp').  This avoids accidental overwriting
      // of useful files.
      strcpy(infilename, filename);
      strcat(infilename, ".inp");

      strcpy(docfilename, filename);
      strcat(docfilename, ".doc");
      strcpy(tempfilename, filename);     strcat(tempfilename, ".tmp");

      strcpy(hdrfilename, filename);
      strcat(hdrfilename, ".hdr");

      strcpy(impfilename, filename);      strcat(impfilename, ".imp");

      strcpy(intfilename, filename);   // This is a documentation file
for developer how want
      strcat(intfilename, ".int");   // to inherit from this class.

      // Set input file for scanning.
```

```
        token.set_file(infilename);
        cout << "Reading from " << infilename << endl;

        // Create a module by parsing the input file.
        Module *m = new Module(&token);

        // Generate the documentation file.
        cout << "Writing to " << docfilename << endl;
        ofstream interface(docfilename);
        m->gen_interface(interface);
        interface.close();
        // Generate the temporary file.      cout << "Writing to " <<
tempfilename << endl;   ofstream temp(tempfilename);  m-
>gen_header(temp);      temp.close();      // Compare the temporary
header file with the current header file.       // Open the previous
header file if there is.            ifstream headeri(hdrfilename);
      if (headeri.fail())           cout << "There is no file named as
" << hdrfilename << endl;     else  {           // Open the temporary
header file.            ifstream tempi(tempfilename);       while
((!tempi.eof()) && (!headeri.eof()) && (!newhead))          {
      // Read the previous header file.
      headeri.getline(bufhdr, BUFLEN);                  // Read the
temporary header file.              tempi.getline(buftemp, BUFLEN);
            if (strcmp(buftemp, bufhdr) != 0 )
      newhead = true;         }           tempi.close();
      headeri.close();  }

      if (newhead)
      {
            // Generate the header file.
            cout << "Writing to " << hdrfilename << endl;
            ofstream header(hdrfilename);
            m->gen_header(header);
            header.close();
      }

      // Generate the implementation file.      cout << "Writing to "
<< impfilename << endl; ofstream impl(impfilename);   m-
>gen_impl(impl);  impl.close();

      // Generate the interface file for developer to inherit.
      cout << "Writing to " << intfilename << endl;
      ofstream interface2(intfilename);
      m->gen_interface2(interface2);
      interface2.close();
}
```

****** commentc.h program ******

```
#ifndef COMMENTC_H
#define COMMENTC_H

#include <fstream.h>
#include "node.h"

class Commentc : public Node
      // A node for a Comment.
{
      public:
        Commentc() : Node() {};
            Commentc (Token *tokptr);
            void gen_interface (ofstream & fout);
            void gen_interface2 (ofstream & fout);
            void gen_header (ofstream & fout);
            void gen_impl (ofstream & fout);
      private:
            char *comment;
};

#endif
```

****** feature.h program ******

```
#ifndef FEATURE_H
#define FEATURE_H

#include <fstream.h>
#include "list.h"
#include "node.h"
#include "library.h"
#include "supplier.h"
#include "filename.h"
#include "parameter.h"

// Class headers for parse tree nodes.

enum inheritn
{
        Privaten,
        Publicn,
        Protectedn,
        Nonen,
};

class Feature : public Node
      // A node for a feature which may be a variable or a function.
{
      public:

            Feature();
            Feature (Token *tokptr, char *clname);

            void gen_interface (ofstream & fout);
            void gen_interface2 (ofstream & fout);
            void gen_header (ofstream & fout);
            void gen_impl (ofstream & fout);

      private:

            bool fun;          // Distinguishes functions/variables.
            bool str;          // Distinguished Star character.
            bool ands;         // Distinguished Ampersand character.
            bool array;      // Didtinguished Array character variables.
            bool constrct;     // Didtinguished class constructor.
            bool destrct;      // Didtinguished class destructor.
            bool comf1;        // Distinguished comment string.
            bool comf2;        // Distinguished comment string.

            char *classname;         // Name of owner class.
            char *type;              // Type of feature.
            char *name;              // Name of feature.
            char *body;              // Body of function.
            char *arrayname;         // Name of array.
            char *arraysize;         // Size of array.

            List *params;            // List of parameters.
            List *commentf1;         // Comment associated with feature.
```

```
            List *commentf2;        // Comment associated with feature.
            inheritn inheritance;   // Inheritance kind of feature.
};

#endif
```

****** filename.h program ******

```
#ifndef FILENAME_H
#define FILENAME_H

#include <fstream.h>
#include "node.h"
#include "module.h"

// Class headers for parse tree nodes.

class Filename : public Node
      // A node for a file name.
{
      public:
        Filename() : Node() {};
            Filename (Token *tokptr);
            void gen_interface (ofstream & fout);
            void gen_interface2 (ofstream & fout);
            void gen_header (ofstream & fout);
            void gen_impl (ofstream & fout);
      private:
            char *fname;
            char *text;
            char *dirname;
};

#endif
```

****** library.h program ******

```
#ifndef LIBRARY_H
#define LIBRARY_H

#include <fstream.h>
#include "node.h"

class Library : public Node
      // A node for a library directive.
{
      public:
        Library() : Node() {};
            Library (Token *tokptr);
            void gen_interface (ofstream & fout);
            void gen_interface2 (ofstream & fout);
            void gen_header (ofstream & fout);
            void gen_impl (ofstream & fout);
      private:
            char *liblist;
            bool comal;
            bool semil;
};

#endif
```

****** list.h program ******

```
#ifndef LIST_H
#define LIST_H

#include "node.h"

// These classes handle lists of nodes.

class Listnode
{
      friend class List;
      private:
            Listnode (Node *p);
            Node *curr;
            Listnode *next;
};

class List
      // An instance is the root of a list.  It has pointers to both
      // ends of the list, so that new nodes can be added at the end.
{
      public:
            List ();
            void append (Node *p);

      // Append a node to the end of the list.
      // The following functions call the corresponding generators
      // for each item in the list.
            void gen_interface (ofstream & fout);
            void gen_interface2 (ofstream & fout);
            void gen_header (ofstream & fout);
            void gen_impl (ofstream & fout);

      private:
            Listnode *first;        // First node in list.
            Listnode *last;         // Last node in list.
};

#endif
```

****** module.h program ******

```
#ifndef MODULE_H
#define MODULE_H

#include <fstream.h>
#include "node.h"
#include "library.h"
#include "supplier.h"
#include "filename.h"
#include "feature.h"
#include "commentc.h"

// Class headers for parse tree nodes.

enum inherit
{
        Private,
        Public,
        Protected,
        None,
};

// A module corresponds to a single class.
class Module : public Node
{
      public:
            Module() : Node() {};
            Module (Token *tokptr);
            void gen_interface (ofstream & fout);
            void gen_interface2 (ofstream & fout);
            void gen_header (ofstream & fout);
            void gen_impl (ofstream & fout);
      private:
            List *libraries;        // List of library directives.
            List *suppliers;        // List of supplier directives.
            List *features;                 // List of features.
            List *publics;                  // List of publics.
            List *friends;                  // List of friends.
            List *protecteds;             // List of protecteds.
            List *privates;               // List of privates.
            List *comments1;              // List of comments;
            List *comments2;              // List of comments;
            List *comments3;              // List of comments;
            List *comments4;              // List of comments;
            List *comments5;              // List of comments;
            char *classname;              // Name of class.
            char *comment;          // Character string for comment.
            char *friendname;             // Friend class name.
            char *inhrname;               // Inheritance class name.
            bool frnd;              // Distinguished friend class.
            inherit inheritance;          // Kind of class Inheritance.
};

#endif
```

****** node.h program ******

```
#ifndef NODE_H
#define NODE_H

#include <fstream.h>
#include "token.h"

// Class headers for parse tree nodes.

class List;

const int FILENAMELEN = 40 ;

class Node
      // Abstract base class for all nodes.
{
      protected:
            Token *tp;          // Pointer to the token.
      public:

            Node () { tp = 0 ;};
      // Every node class must implement the following functions.
      // They generate text for the interface (or documentation)
      // file, header file, and implementation file, respectively.

            virtual void gen_interface (ofstream & fout) {};
            virtual void gen_interface2 (ofstream & fout) {};
            virtual void gen_header (ofstream & fout) {};
            virtual void gen_impl (ofstream & fout) {};
};

#endif
```

****** parameter.h program ******

```
#ifndef PARAMETER_H
#define PARAMETER_H

#include <fstream.h>
#include "node.h"
#include "module.h"
#include "library.h"
#include "supplier.h"
#include "feature.h"
#include "filename.h"
#include "token.h"
#include "list.h"

// Class headers for parse tree nodes.


class Parameter : public Node
      // A node for a single parameter.
{
      public:
                Parameter();
            Parameter (Token *tokptr);
            void gen_interface (ofstream & fout);
            void gen_interface2 (ofstream & fout);
            void gen_header (ofstream & fout);
            void gen_impl (ofstream & fout);
      private:
            bool str;
            bool bool_separator;
            bool ands;
            char *type;                         // Type of parameter.
            char *name;                         // Name of parameter.
};

#endif
```

****** supplier.h program ******

```
#ifndef SUPPLIER_H
#define SUPPLIER_H

#include <fstream.h>
#include "node.h"

class Supplier : public Node
      // A node for a supplier directive.
{
      public:
        Supplier() : Node() {};
            Supplier (Token *tokptr);
            void gen_interface (ofstream & fout);
            void gen_interface2 (ofstream & fout);
            void gen_header (ofstream & fout);
            void gen_impl (ofstream & fout);

      private:
            char *suplist;
            bool comas;
            bool semis;
};

#endif
```

****** token.h program ******

```
#ifndef TOKEN_H
#define TOKEN_H

#include <fstream.h>
#include <iostream.h>

// Header file for scanner.

// Define buffer sizes.
const int IDENLEN = 50;
const int BUFLEN = 120;
const int BODYLEN = 4000;
const int COMLEN = 120;

// Declare kinds of token
enum KIND
{
      IDENTIFIER,       // Identifier that is not a keyword
      INCLUDE,          // "include"
      LIBRARY,          // "library"
      SUPPLIER,         // "supplier"
      CLASS,            // "class"
      PUBLIC,           // "public"
      PRIVATE,          // "private"
      BODY,             // function body between '{' and '}'
      COMMENT,          // comment
      NUMSIGN,          // "#"
      COMMA,            // ","
      DOT,              // "."
      QUOTE,            // '"'
      STAR,             // "*"
      COLON,            // ":"
      VIRTUAL,          // "virtual"
      PROTECTED,        // "protected"
      CHAR,             // "char"
      INT,              // "int"
      VOID,             // "void"
      UNDERSCORE,       // "_"
      AND,              // "&"
      OR,               // "|"
      PERCENT,          // "%"
      PLUS,             // "+"
      MINUS,            // "-"
      DOLLAR,           // "$"
      LONG,             // "long"
      RETURN,           // "return"
      EQUAL,            // "="
      CONST,            // "const"
      TILDA,            // "~"
      DOUBLE,           // "double"
      STATIC,           // "static"
      TYPEDEF,          // "typedef"
      FRIEND,           // "friend"
      SEMICOLON,        // ";"
      LEFT_BRAC,        // "{"
```

```
        RIGHT_BRAC,         // "}"
        LEFT_PAREN,         // "("
        RIGHT_PAREN,        // ")"
        LEFT_ANGLE,         // "<"
        RIGHT_ANGLE,        // ">"
        LEFT_BRAK,          // "["
        RIGHT_BRAK,         // "]"
        SLASH,              // "/"
        BACKSLASH,          // "\"

        END_FILE,           // end of file
        BAD,                // bad token (indicates an error)
        CONSTRUCTOR,
        DESTRUCTOR
};

class Token
{
        public:

                void set_file(char *infilename); // assign a file for input
                void newline ();        // read one line from the input file
                void skip ();          // skip blanks, tabs, and line breaks
                void read ();                   // read one token
                void read_body ();      // read the body of a function
                void error (char *text);        // report an error
                void match (KIND kd);   // match a specific token kind
                char tokenkind (Token *tk);
                char* tokeniden (Token *tk);
                char* tokenbody (Token *tk);
                char* tokencomment (Token *tk);

        private:

                KIND kind;                      // kind of token
                char iden[IDENLEN];  // buffer for keywords and identifiers
                char buffer[BUFLEN];            // input file line buffer
                char body[BODYLEN];             // text of function body
                char comment[COMLEN];           // text of comment

                ifstream fin;     // input file from which we read tokens
                int line;                       // current line number
                char *p;                        // pointer to line buffer
                bool overflow;
};

#endif
```

****** canonical file (input) example, test.inp ******

```
// Includes stdio, stdlib and iostream libraries

library stdio, stdlib, iostream;  // Libraries files name list

// supplier means client's header file
supplier myclass, yourclass, theirclass; // test supplier comment

class Test : public T1  // test class comment
      // A class for testing things

friend class T3;   // for test comment in friend line
// T3 is a friend class

public constructor Test(int k) // constructor comment
{
      k = 5;  // test constructor comment
// test constructor comment
}

public destructor Test();  // test A0
// test AA
protected char pr2; // test AB
// test AC

private int widget_count; // test AD
      // Counter for widgets
char cc;    // test AAA
// test AAB
public int x;     // test AX
public char s[z]; // test AS
public char *tr;  // test AT
public char &nd; // test AN
// test ANN
public void f()   // A pointless function
{
      if (flag) // test 1
      {
            return 0; // test 2
// test 3
      }
      // { test braces }}
      else  // test 4
      {
            return 1;   // test 5
// test 6
      }     // test 7
      char bb == " {{ test braces } ";    // test 8
}

private double variable_with_long_name;   // test 7

private char *strtest;  // test 8

private char &andsgn;   // test 9
// test 10
```

```
private char testarray[arraylen];

protected char prot;

private float g (int n, float x, char *ts, char arry[arrylen], char
&amp;amp)
      // Another pointless function
{
      x = sqrt(n);      // test 11
}
```

****** interface (documentation) file example, test.doc ******

```
// Includes stdio, stdlib and iostream libraries

library stdio, stdlib, iostream;

// Libraries files name list
// supplier means client's header file

supplier myclass, yourclass, theirclass;

// test supplier comment
//    #include <c:/yousefi/university/stdio.h>

class Test :  public T1
// test class comment
// A class for testing things

{
      friend class: T3;

// for test comment in friend line
// T3 is a friend class

      public:
            CONSTRUCTOR Test(int k);
            // constructor comment

            DESTRUCTOR Test();
            // test A0
// test AA

            int x;
            // test AX

            char s[z];
            // test AS

            char *tr;
            // test AT

            char &nd;
            // test AN
// test ANN

            void f();
            // A pointless function

};
```

****** interface2 (documentation2) file (output) example, test.int ******

```
// Includes stdio, stdlib and iostream libraries
library stdio, stdlib, iostream;

// Libraries files name list
// supplier means client's header file

supplier myclass, yourclass, theirclass;

// test supplier comment
//   #include <c:/yousefi/university/stdio.h>

class Test :  public TI
// test class comment
// A class for testing things

{
      friend class: T3;
// for test comment in friend line
// T3 is a friend class

      public:

            CONSTRUCTOR Test(int k);
            // constructor comment

            DESTRUCTOR Test();
            // test A0
// test AA

            int x;
            // test AX

            char s[z];
            // test AS

            char *tr;
            // test AT

            char &nd;
            // test AN
// test ANN

            void f();
            // A pointless function
      protected:

            char pr2;
            // test AB
// test AC

            char prot;
};
```

****** header file (output) example, test.hdr ******

```
#ifndef     TEST_H
#define     TEST_H

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

#include "myclass.h"
#include "yourclass.h"
#include "theirclass.h"

class Test : public T1

{
      friend class T3;
      public:
            Test(int k);
            ~Test();
            int x;
            char s[z];
            char *tr;
            char &nd;
            void f();

      protected:
            char pr2;
            char prot;

      private:
            int widget_count;
            char cc;
            double variable_with_long_name;
            char *strtest;
            char &andsgn;
            char testarray[arraylen];
            float g(int n, float x, char *ts, char arry[arrylen], char
&amp);

};
#endif
```

****** implementation file (output) example, test.imp ******

```
#ifndef     TEST_H
#define     TEST_H

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

#include "myclass.h"
#include "yourclass.h"
#include "theirclass.h"

Test :: Test(int k)
{
      k = 5;  // test constructor comment
// test constructor comment
}

Test :: ~Test();

void Test :: f()
{
      if (flag) // test 1
      {
            return 0; // test 2
// test 3
      }
      // { test braces }}
      else  // test 4
      {
            return 1;   // test 5
// test 6
      }     // test 7
      char bb == " {{ test braces } ";     // test 8
}

float Test :: g(int n, float x, char *ts, char arry[arrylen], char
&amp)
{
      x = sqrt(n);      // test 11
}
```
