1998

Extension of object-oriented use case-driven approach software engineering.

Frank. An
University of Windsor

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®

EXTENSION OF OBJECT ORIENTED USE CASE DRIVEN APPROACH SOFTWARE ENGINEERING

by

Frank An

A Thesis

Submitted to the College of Graduate Studies and Research

through the School of Computer Science

in Partial Fulfillment of the Requirements for the

Degree of Master of Science

at the University of Windsor

Windsor, Ontario, Canada

1998

Frank An 1998

ABSTRACT

Use case driven approach is a method of object oriented software engineering (OOSE) developed by Jacobson *et al.* (1992). The requirement model of the use case driven approach software system is defined by a collection of use cases, problem domain descriptions and user interfaces. Extension is an important type of association between use cases. Extension means that a use case that represents the major or basic course of events can be naturally extended with one or several other use cases that represent rare or exceptional courses of events. A difficulty for OOSE to transform the use cases with extension associations into the design and implementation model of the system is that most programming languages, such as C++, Smalltalk and Java, do not have features to support the extension association. The current implementation of an extension association still relies on the basic course use case to initiate the extension use cases.

This thesis presents a programming technique to overcome the difficulty described above. It supports the integration of a major course use case and its extension use cases. Based on this technique, a mechanism *extension construct invocation* is proposed to directly support the extension association between use cases. The *extension statement* and *probing clause* allow an extension use case to automatically respond to any extension request from a basic course use case. This mechanism reduces the responsibility of a method that represents a basic course use case with extension statements. The probing clause added to a method that represents an extension use case can automatically respond to any extension request from a basic course use case.

To my parents, Dr. Chang-Fa An and Suhua Yang

My brothers, Edward An and Angus An

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my supervisor, Dr. Liwu Li, for his instructive advice. Without his help and encouragement, this thesis could not have been completed. I greatly appreciate Dr. Young G. Park from the School of Computer Science, University of Windsor, for his constructive advice and suggestions. I am particularly grateful to Dr. Ronald Barron from Mathematics and Statistics, University of Windsor, for his valuable comments and suggestions.

TABLE OF CONTENTS

LIST OF FIGURES

Page

Chapter 1

INTRODUCTION

Object Oriented Software Engineering (OOSE) developed by Jacobson *et al.* (1992) is one of the major object oriented software development approaches. This method is a use case driven approach and it based on the use cases of a software system. The use cases are derived from the user's requirements and define the functionality of the software system. The use cases are used to derive an analysis model, which is independent of the implementation environment and refined to a design model. The design model adapts the analysis model for implementation environment and is used to implement the software system. The use cases are indispensable to testing the model as well. Thus, the use cases are traceable for all models produced in an OOSE process. OOSE provides seamless transitions among various phases of the software life-cycle including system analysis, design, implementation and testing.

Use case is a complete course of interactions between a particular type of user and a software system. It represents what users can do with the system. Use case is a unique feature of OOSE, not available in other object oriented analysis and design approaches such as those developed by Rumbaugh *et al.* (1991) and Booch (1994). A set of use cases of a software system completely describes the functionality of the software system.

Extension association is an important concept of OOSE. As indicated by Jacobson *et al.* (1992), extension specifies how one use case inserts itself into

another use case to extend the functionality. Thus, changes in and additions to the functionality of a use case can be made more easily.

According to the description of use case, a use case is independent of any other use cases that are to be inserted into it. It does not mention or pay attention to the extension use case. An obvious benefit of using the extension association is the simplicity of the description of the extended use cases. It also implies flexibility and efficiency in packaging use cases for different applications, and facilitates the reuse of extension use cases.

1.1 Goals of the Thesis

As indicated by Jacobson *et al.* (1992), common programming languages do not support extension association directly. Ordinary programming languages have no construct by which we can express extension association. None of the common object oriented programming languages, such as C++ (Ellis and Stroustrup (1992) and Stroustrup (1991)), Smalltalk (Goldberg and Robson (1983, 1989)) and Java (Arnold and Gosling (1996) and Gosling *et al.* (1996)), allows extension association.

The main goal of this work is to provide a carefully designed programming technique to implement extension association of the use case driven approach. The implementation of C++ program includes three additional goals:

- Efficiency: The goal is to provide easily assembled components, which impose minimal runtime and memory overhead, and are more reliable than hand-built mechanisms.

- Ease of Use: A clear and consistent organization should make it easy to identify and select appropriate forms of extension association. Providing

nearly independent parts simplifies combining components to create extension constructor.

- Extendibility: It is possible to add new features and more mechanisms, such as using inheritance. The changes in syntax and semantics must be easy to implement.

1.2 Scope of the Project

In OOSE, use cases are eventually transformed into different types of objects defined in a programming language. In this project, we propose constructs for programming languages to support the expression of extension association in programs. The constructs are described for both procedural and object oriented programming languages. The proposal can be applied to any procedural language that uses modules to organize procedures. We present approaches to implement the constructs for the object oriented programming language C++. The approaches convert an extension construct to an equivalent program that does not contain the constructs.

The technique introduced in this work is a general technique. A programmer who lacks knowledge of use case or any object oriented analysis and design approach can use this technique effectively. The technique can be used to support the so-called "software plug-in". By including a new module or class into a compilation unit, the technique allows the functionality of the new module or class to be inserted into existing modules or classes without having to modify the existing modules or classes. Therefore, the functionality of an existing program can be extended with new modules and classes.

1.3 Overview of the Thesis

This thesis is organized as follows. In chapter 2, we present a survey of software engineering, object oriented programming languages and object oriented software engineering with use case driven approach. In chapter 3, we introduce the notion of extension construct for programming languages. Extensions can be used in procedures and functions corresponding to method of classes in C++. We formalize the extension constructs by specifying an operational semantics for extensions between C++ functions. In chapter 4, we describe a programming technique for translating classes that use extension constructs to classes that contain no extension construct. We also present a technique that uses hash table to store extension parameters to improve the efficiency of the resulting program. In chapter 5, we summarize the results and conclusions of this work.

Chapter 2

LITERATURE SURVEY

2.1 Software Engineering and Software Crisis

In the 1960s, with the great advances in hardware and software techniques, there were lots of difficult technical and managerial problems in the development of large-scale software systems. Many failures of software projects let to a *software crisis* (Ghezzi *et al.* 1991).

Software engineering is a method to overcome the software crisis. Parnas (1987) has defined software engineering as "multi-person construction of multi-version software." Software engineering deals with the building of software systems which are so large or so complex that they are built by a team or teams of engineers. The software life-cycle model is an important model to practical developments in software engineering.

There are two goals of software specification requirement in software engineering — "it does the right thing" and "it does the thing right". In the real world, it is unfortunately true that it is impossible to guarantee whether or not a system perfectly satisfies this requirement because of some underlying assumptions. These assumptions cause *residual doubt* in software engineering. Residual doubt refers to whether an implementation truly represents its specification or not. In recent years, object oriented analysis and design, which thinks about problems using models organized around real-world concepts, has become a widely used method in the software industry to develop software systems and overcome residual doubt.

2.2 Software Life-Cycle Model

In most discussions of software development methods, techniques are primarily based on a conventional view of the software life-cycle. It covers both traditional structured methods and formal methods.

From the technical development viewpoint of software development process models, five development stages can be identified:

- Requirement analysis
- System specification
- Architectural design
- Detailed design
- Implementation

From the project management viewpoint of software development process models, the waterfall model is commonly used. The waterfall model (McDermid and John 1993) was derived in late 1960s (Figure 2.1).

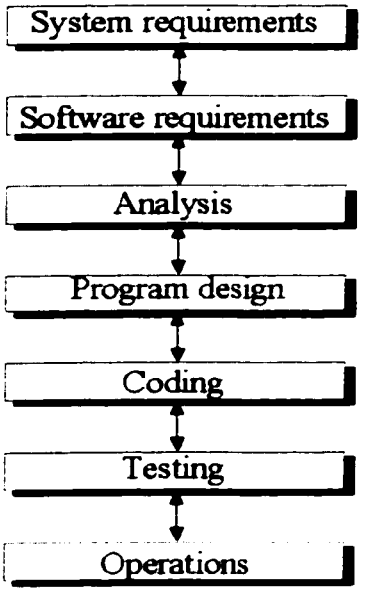

Figure 2.1 The Waterfall Model

There are three methods of software development – conventional, formal and object oriented.

(1) In conventional development, main stages of the development process are as follows.

- Requirement analysis (Stokes 1993) is typically the first stage of the development process. It builds a model of a real-world situation showing its important properties. It has long been recognized that this stage is a very difficult step. Mistakes made in the requirement analysis stage may have a profound effect on the usability of the eventual system.

- System specification (Coleman and Gallimore 1987) is the next stage of the software system development process. It is concerned with definition of the system architecture, or "grand plan" for the system. It identifies the major system components and their inter-relationships.

- Architectural design (Buxton and McDermid 1993) is the kernel in the system development process. It describes the system interfaces, functionality and structure as the designers intend to implement them. Errors are often made at this stage but not caught until the implementation stage. Like requirement analysis, errors made at this stage can be very expensive and very difficult to correct later.

- Implementation (Wichmann 1993) is the final development stage that is concerned with detailed program design and the production of the program source code. Architectural design identified modules that can be implemented by individual software engineers. Implementation is concerned with the production of these individual modules.

(2) Formal methods are methods for developing software while also facing up to make difficult decisions of design. They are methods for recording those decisions once made. They help us at each stage by asking what decisions have been made and what decisions have been deferred, and perhaps encourage us to record the reasons for making or deferring decisions. Formal methods provide a development organization with a complete record of the design decisions that has led to the product to be the way it is, and this record is a valuable asset in the continuing life of a product.

(3) Object oriented development method is a relatively new approach of software engineering. It is achieving a wide acceptance as a standard basic technology

within the software industry. This new technique changes the nature of software design from being an art or a craftsmanship to being an industrial process. Software products can be developed in a way that is similar to hardware products, such as "plug-in" technique.

The object oriented method is based on using objects as abstractions of the real word. The term object oriented development (Jacobson *et al.* 1992) has been used to cover various stages of the software life-cycle, from system analysis through design to implementation. Object oriented means that the software is organized as a collection of discrete objects that incorporate both data structure and behavior. The philosophy of object orientation is to model the real world entities with objects and create reusable, extendible and robust computer programs.

Object oriented programming is a programming methodology that is practical and useful for modular design and software reuse. Most object oriented programming languages support "data abstraction". This is the ability to define new types of objects whose behavior is defined abstractly, without reference to implementation details such as the data structure and methods used to represent the objects. Data abstraction prevents an object from being manipulated except only by its defined external operations.

Object oriented programming languages (OOPLs) have three important features that distinguish them from traditional programming languages: encapsulation, polymorphism and inheritance. *Encapsulation* is the ability of an object to hide internal information from being accessed outside. It also implies the ability to bind data with operations on the data. *Polymorphism* allows for generic operations to take on different forms in different classes via dynamic binding, where the method to be executed is identified at runtime. *Inheritance* is the ability of an object to derive its features from other objects.

2.3 Object Oriented Design

There are three main approaches of object oriented design:

- Object Modelling Technique (OMT) – Rumbaugh *et al.* (1991)
- Object Oriented Analysis and Design – Booch (1994)
- Object Oriented Software Engineering (OOSE) – Jacobson *et al.* (1992)

2.3.1 Object Modeling Technique (OMT)

The OMT methodology (Rumbaugh *et al.* 1991) was developed at General Electric Research and Development Center (GE R&D) in New York. The software life-cycle model supported by this method is given in the following stages:

- Analysis builds a model of the real-world situation showing its important properties. The objects in the model should be application domain model concepts but not computer implementation concepts such as data structures.
- System design makes high level decisions about the overall architecture.
- Object design builds a design model based on the analysis model and contains implementation details. The focus of object design are the data structures and algorithms, which are needed to implement classes. Both the application domain objects and the computer domain objects are described

using the same object oriented concepts and notation, although they exist on different conceptual planes.

- In the implementation stage, the object classes and relationships are finally translated into a particular programming language, such as C++, database and hardware implementation.

In the OMT design, there are three models to answer what, how and when to change.

- *Object model* describes the static structure of the objects in a system and their relationships.

- *Dynamic model* describes the interactions among objects in the system.

- *Functional model* describes the data transformations of the system.

2.3.2 Booch Approach

Object oriented analysis and design is the method that leads us to an object oriented decomposition. Object oriented design defines a notation and process for constructing complex software systems and offers a rich set of logical and physical models with which we may reason about different aspects of the system under consideration.

In the Booch approach, an object has state, behavior and identity. The *state* of an object encompasses all of the properties of the object plus the current values of each of these properties. *Behavior* is how an object acts and reacts in terms of its state changes and message passing. *Identity* is the property of an object that distinguishes it from all other objects.

The notation for object oriented development includes four basic diagrams (class diagram, object diagram, module diagram and process diagram) and two supplementary diagrams (state translation diagram and interaction diagram).

Successful projects are usually characterized by a strong architectural vision and a well-managed iterative and incremental development life-cycle. A completely rational design process is not possible, but can be faked by reconciling the micro and macro process of development.

The *micro process* represents daily activities of the development team. It is described in the following steps.

- Identify the classes and objects.
- Identify the semantics of these classes and objects.
- Identify the relationships among these classes and objects.
- Implement these classes and objects.

The *macro process* defines a number of measurable products and activities for managing risk. Macro process is described in the following steps.

- Conceptualization, which establishes the core requirements for the system.
- Analysis, which provides a model of the system's behavior.
- Design, which creates an architecture for the implementation and establishes common tactical policies.

- Evolution, which uses successive refinement to ultimately lead to the production system.

- Maintenance, which is essentially the management of post-delivery evolution.

2.3.3 OOSE – Use Case Driven Approach

The software life-cycle model supported by this method is given in the following OOSE model architecture:

1) The requirement model aims to capture the functional requirements.

   - A use case model

   This model defines what exists outside the system (*actors*) and what should be performed by the system (*use case*).

   - Interface descriptions

   This is a man-machine interface (MMI). Which can be used to simulate the use cases as they will appear to the users before thinking of how to realize them.

   - A problem domain model

   This presents a logical view of the system. This model is a good tool to communicate with the system.

2) The analysis model aims to give the system a robust and changeable object structure. There are three object types used to structure the system in the analysis model.

- *The interface objects* are used to model functionality that is directly dependent on the system interfaces.

- *The entity objects* should model information that the system should manage for a long time, and the behavior tied to this information.

- *The control objects* should model functionality that is transaction-oriented for a specific use case and that should be kept together for maintenance purpose.

3) The design model aims to accept and refine the object structure to the current implementation environment.

4) The implementation model aims to implement the system.

5) The test model aims to verify the system.

In OOSE, the user's requirements of a software system can be used to create a requirement model. The main component of the requirement model is the use case model.

A use case model uses *actors* to abstract the roles played by users of a software system. The actors represent anything that interacts with the system. They represent everything that needs to exchange information with the system. Since the actors represent what is outside the system, we do not describe them in detail.

The actor initiates the *use case*. An instance of an actor carries out a number of different operations on the system. When a user uses the system, the system will perform a behaviorally related sequence of transactions with the system. We call such a special sequence of transactions a *use case*. The system uses a use case to

describe what should be performed by the software system for an actor. The use case progresses through interactions and information exchanges between the actor and system.

In order to obtain a picture of how the object fits into the system, different use cases can be described in which the object takes part and communicates with other objects. In this way, we can fully describe the object's surroundings and what the other objects expect of this object.

Use cases are the focus of the use case driven approach in OOSE. They formulate the needs of the users of a software system. They are used to deduce an analysis model, which is a logical, stable and implementation-environment independent model of the software system. The analysis model uses different types of objects to represent use cases. The functionality of the software system is dispatched to objects of the analysis model. A use case may be implemented with one or several objects. The functions or methods in the objects realize the functionality of the use case.

The analysis model is then refined to a design model, which considers the complexity of the implementation environment. The system will be implemented from the design model. The use cases can be traced from the analysis model, from the design model and from the implementation of the software system.

Designing in this way, the system model will be use case driven. To change the system behavior, we remodel the appropriate actors and use cases. The whole system architecture will be controlled by what the users wish to do with the system. Since we have trace ability through all models, it is possible modify the system to meet new requirements. The users are asked what they want to change (which use case) and the developer sees directly where these changes should be

made in the other models. The use case model will control the formation of all other models.

An actor is regarded as a class and user as an instance of this class. A use case is an object that has state and behavior.

2.3.4 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is the successor to the wave of object oriented analysis and design (OOA&D) methods that appeared in the late 1980s and early 1990s. It directly unifies the methods of Booch (1994), Rumbaugh *et al.* (1991) and Jacobson *et al.* (1992), but it is wider than each of these.

UML collects ideas about object oriented analysis and design from Booch(1994). It uses notations and semantics from Rumbaugh *et al.* (1991). The process of using UML on an object oriented project comes from Jacobson *et al.* (1992). UML lies at the center of a standardization process with the Object Management Group (OMG). In 1997, various organizations submitted proposals for a methods standard to facilitate the interchange of models. These proposals focus on a meta-model and an optional notation. Rational company released version 1.1 of the UML document as their proposal to the OMG in September 1997. It is expected to be the standard modeling language in the future.

UML is called a modeling language, but not a method. Most methods consist of both a modeling language and a process. The modeling language is the notation that methods use to express designs. The process is the advice of decisions used by notations on what steps to take in doing a design. The UML specification consists of two interrelated parts:

- UML Semantics. This is a meta-model that specifies the abstract syntax and semantics of UML object modeling concepts. It includes four parts as follows:

  1) Language architecture and language formalism.

  2) Foundation package, which is decomposed into several subpackages – core, auxiliary elements, extension mechanisms and data types.

  3) Behavioral elements package, which consists of four lower-level packages – common behavior, collaborations, use cases and state machines.

  4) General mechanisms package.

- UML Notation. This is a graphic notation for the visual representation of the UML semantics. The notation is the graphical images you see in models. It is the syntax of the modeling language. UML defines several diagrams — static structure diagrams, use case diagrams, sequence diagrams, collaboration diagrams, state chart diagrams, active diagrams and implementation diagrams.

In UML, there are three relationships among use cases — communicate, extend and use. *Communicate* relationship means that the participation of an actor in a use case. *Extend* relationship from use case A to use case B indicates that an instance of use case B may include the behavior specified by A. *Use* relationship from use case A to use case B indicates that an instance of the use case A will also include the behavior as specified by B. Extend relation is just a concept in OOSE by Jacobson *et al.* (1992). In UML diagram, it is shown by a generalization arrow from the use case providing the extension to the base use case. This arrow is labeled with the stereotype <<extends>>.

Chapter 3

DESIGN OF EXTENSION CONSTRUCTS

3.1 Use Case Driven Software Development with Extension

Extension association is an important concept that is used to compose use case descriptions. It specifies how one use case description may be inserted into, and thus extend to, another use case description.

An automated recycling machine is used as an example (Jacobson *et al.* (1992)) to illustrate the notions of actors and use cases. There are two actors: Customer and Operator. Customer deposits used cans, bottles and crates into the recycling machine, which recognizes the returned items, counts them, computes their values and prints a receipt at the request of the Customer to complete the use case. This is a use case called "Returning items". It is a major use case. There are two extension use cases for this major use case. One deals with the exceptional situation that a returned item is stuck in the machine. The other deals with running out of receipt paper. The major use case does not pay attention to the extension use cases. The extension use cases are still complete courses of transactions. They specify the functionality of the recycling machine for the exceptional cases.

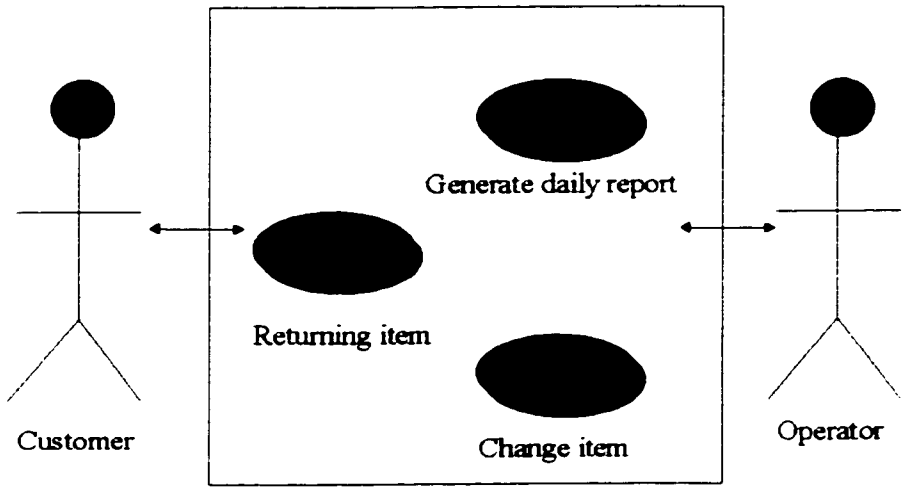

Figure 3.1.1 Recycling Machine
(from Jacobson *et al.* (1992))

In a use case model, use cases are described in a natural language such as English. A use case describes the interactions between a system and an actor. For example, two of the above three use cases can be described naturally in English as follows.

Returning item: This use case is started by Customer when he wants to return cans or bottles. With each item that Customer places in the recycling machine, the system will increase the received number of items from Customer as well as the daily total of this particular type. When Customer has deposited all his items, he will press the receipt button to get a receipt on which the returned items have been printed as well as the total return sum.

Item is stuck: When an item gets stuck the alarm is activated to call the Operator. When the Operator has removed the stuck item he resets the alarm and the Customer can continue to

return items. The Customer's total so far is still valid. The Customer does not get credit for the stuck item.

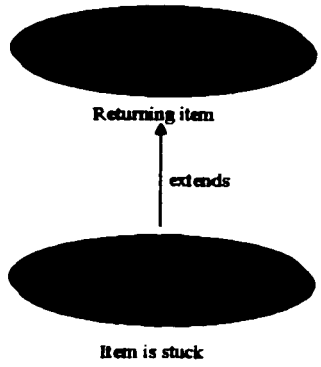

Figure 3.1.2 Use Case Extension Association

As indicated by Jacobson *et al.* (1992), an extension use case should state the insertion position in the extended use case as exactly as possible. Assume that the extended use case has been started. When it runs to a position that the extension use case specifies as insertion position, the extension use case is started and the extended use case is interrupted. Upon the completion of the extension use case, the interrupted use case resumes its execution.

In the early stages of software system development, it may not be easy to specify the exact position in an extended use case. For example, when the major use case for the recycling machine system is described in a natural language, the

functionality is described in an abstract way. When a use case is implemented with one or several classes in the design model of the system, code or pseudo-code is used to describe operations. In the design model or in the implementation model, the exact position can easily be identified for an extension use case that is represented by some method in some class.

From the above discussion, we can see the need for the expression of a position in a procedure or method to support extension use case insertion. The position is used in the implementation of an extension use case. The specification of an extension use case explicitly mentions the extended/major use case. It knows the extended use case, but may or may not know the extension use case. There is obviously a need for a construct for representing an extended use case in the implementation of an extension use case.

Another emphasis of OOSE is the reusability of use cases. An extension use case may be inserted at various positions in several use cases. A use case may be extended by several other use cases. The complexity of extension relation demands flexible constructs to express extension associations.

A problem addressed in this thesis is how to express the extension relation between use cases. The expression should allow a programmer to specify an extended use case in a use case. It should allow identifying insertion positions in an extended use case and using the positions in an extension use case.

3.2 Syntax of Extension Constructs

We introduce two new keywords, *extension* and *probing*, for any computer language that supports the notions of procedure and module that correspond to the method and class in an object oriented programming language. Keyword *extension* is used to declare locations in a program where the program can be extended.

The keyword *extension* is used to introduce an extension statement. The keyword *probing* is used to declare procedures or methods that can be used to extend a program at locations declared with extension statements.

We will use the context-free grammar, BNF notation, to describe the syntax of the extension statement and probing clause. The meta-language BNF (Aho *et al.* 1986), refers to either Backus-Naur Form or Backus Normal Form. Formulated originally by John Backus (1981), who was the father of FORTRAN, it was first made popular when it was used to describe the syntax of ALGOL 60. In BNF, the symbols "<" and ">" represent syntactic classes of elements. The symbol "::=" separates the syntactic class <classname> from its definition. The nonterminal symbols V_N are always parenthesized with the meta-symbols "<" and ">". A terminal symbol V_T is a character or a string of characters from the alphabet. It is convenient to use a meta-linguistic symbol "|", read as "or", to separate the right parts of productions that have a common left part.

We define a grammar more formally as a 4-tuple $G = (V_N, V_T, S, P)$ where V_N and V_T are disjoint sets of nonterminal and terminal symbols. S is the distinguished symbol of V_N, commonly called the *goal* or starting symbol. P is a finite set of productions. The set $V = V_T \cup V_N$ is called the *vocabulary* of the grammar.

The concept of recursion is very applicable to grammars. An example of a grammar which defines a simple language, string of numbers, recursively is as follows.

```
G = (VN, VT, S, P)
where VN= {<digit>,<no>,<number>}
      VT= {0,1,2,3,4,5,6,7,8,9}
      S = { <number> }
      P = { 1  <number> ::= <no>
            2  <no>     ::= <digit> | <no> <digit>
```

```
    3   <digit>   ::= 0|1|2|3|4|5|6|7|8|9
}
```

Let the braces "{" and "}" be additions to the meta-symbol set of the basic BNF meta-language. We define {z} to mean 0 or more occurrences of z. The syntax of a term in an arithmetic expression will be:

<term> ::= <factor> { * <factor> }

3.2.1 Syntax of Extension Statement

An extension statement may convey optional information to describe the probing procedures or methods that can be inserted at the location of the statement. The syntax of an extension statement is

extension (pC:: pM: pL) eF: eL;

where extension is the keyword. We assume that the extension statement is in a class eC with procedure or method named eM. Like statements in C, C++ or Java, an extension statement is terminated with a semicolon ";". The above statement is composed of optional parts pC::, pM:, pL, eF: and eL introducing extension parameters pC, pM, pL, eF and eL. We use symbols "::" and ":" to separate the parts. The specific information carried by the extension statement depends on the parameters pC, pM, pL, eF and eL, which are interpreted as follows.

- pC is a class name if part pC:: is declared in the extension statement. The name may or may not be the same as the module or class eC that contains the extension statement. The parameter pC is called the *probing class* of the extension statement.

- pM is the name of a procedure or method if part pM: is declared. Method name pM may be defined in several different modules or classes. The procedures or methods named pM may or may not include the extension statement. The parameter pM is called the *probing method* of the extension statement.

- pL is a nonnegative integer such as 0, 3, 15, ... if pL is declared. It should be a label of a probing procedure or method, which will be defined shortly. The parameter pL is called the *probing label* of the extension statement.

- eF is a boolean variable declared in the same class or module that includes the extension statement if the part eF: is declared in the extension statement. The parameter eF is called the *extension condition* of the extension statement.

- eL is a nonnegative integer such as 0, 3, 15, ... if the part eL is declared. The parameter eL is called the *extension label* or simply the *label* of the extension statement.

All the above five parameters are optional. There are 32 (2^5) kinds of extension statements which include an extension statement without any parameter. The module or class eC and procedure or method eM, which include the extension statement, can be found by scanning the code. Thus, from an extension statement, we can identify seven pieces of information: eC, eM, eL, eF, pC, pM and pL. The diagram of extension statement is shown in Figure 3.2.

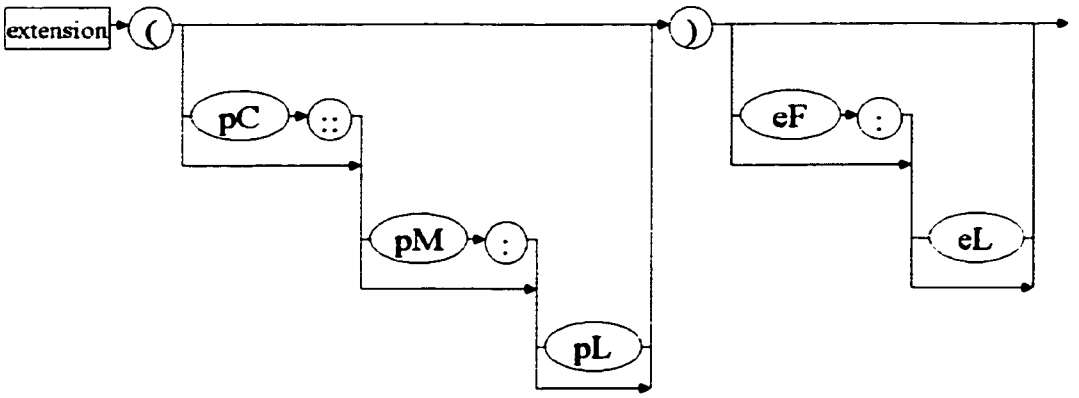

Figure 3.2 Syntax of Extension Statement

The grammar of the extension statement is as follows:

G_1: Grammar of identifier.
G_2: Grammar of number.
G_3: Grammar of expression.
G_4: Grammar of boolean expression.
G_5: Grammar of extension.

Grammar of identifier:
$G_1 = (V_N, V_T, S, P)$
where V_N = {<identifier>,<ident>,<letter>,<others>}
V_T = {A,B,...,Z,0,1,...,9,_,#,$}
S = <identifier>
P = {1 <identifier> ::= <ident>
2 <ident> ::= <letter> {<letter>|<others>}[30]
3 <letter>::= A|B|C| ... |Z|#|$
4 <others>::= 0|1|2| ... |9|_}

Grammar of number:
$G_2 = (V_N, V_T, S, P)$
where V_N = {<digit>, <no>, <number> }
V_T = {0,1,2,...,9 }
S = {<number>}
P = {1 <number>::= <no>
2 <no> ::= <digit> | <no> <digit>
3 <digit> ::= 0|1|2| ... |9 }

Grammar of expression:
$G_3 = (V_N, V_T, S, P)$
where V_N = { <expression>,<expr>,<term>,<factor>,
<number>,<operator>,<function_return> }
V_T = { 0,1,2, ... ,9,+,-,*,/ }
S = <expression>
P = {1<expression>::=<expr>|<function_return>
2<expr>::=<term>|<expr>+<term>|<expr>-<term>
3<term>::=<factor>|<term>*<factor>|
<term>/<factor>
4<factor>::=<digit>|(<expr>) }

Grammar of boolean expression:
$G_4 = (V_N, V_T, S, P)$
where V_N = {<boolean_expression>}
V_T = {TRUE,FALSE}
S = <boolean_expression>
P = {1 <boolean_expression>::=<TRUE>|<FALSE> }

Grammar of extension:
$G_5 = (V_N, V_T, S, P)$
where V_N = {<extension-statement>,<extension>,
<probing_ps>,<extension_ps>,
<identifier1>,<identifier2>,<expression1>,
<expression2>,<boolean_expression>}
V_T = {**extension**,"(",":",":",")",";"}
S = <extension-statement>
P = {1 <extension-statement>::=<extension>
2 <extension>::=**extension**"("<probing_ps>")"
<extension_ps> ";"
3 <probing_ps>::=
<identifier1>"::"|<identifier2>":"|<expression1>

```
        5 <extension_ps>::=
<boolean_expression>":"|<expression2>
        6 <identifier1>::=probing_class_name
        7 <identifier2>::=probing_method_name
        8 <expression1>::=probing_label_expression
        9 <expression2>::=extension_label_expression
```

The following example uses several statements to illustrate the extension statement syntax.

Example 3.2.1 The following are extension statements:

```
extension( );
extension(Alarm::item_is_stuck:);
extension(Alarm::item_is_stuck:5) extension_on: 1;
```

The first statement do not declare any of the optional parameters. It simply indicates a location in a program where extension use cases can be inserted. The second statement provides only a probing class name *Alarm* and the probing method name *item_is_stuck*. The last statement contains all the optional parameters. The probing class for the extension statement is *Alarm*. The probing method name is *item_is_stuck*. The probing label is 5. The extension condition is *extension_on*. The extension label is 1. The statement permits a probing method with name *item_is_stuck* in class *Alarm* to extend the position of the extension statement if boolean condition *extension_on* is *true* and the probing method defines label 5. The label of this extension statement is 1, which is used by probing methods.

3.2.2 Syntax of Probing Clause

Keyword *probing* is used in a program to declare procedures or methods that can extend the program at locations declared with compatible extension statements. A procedure or method that is declared with keyword *probing* is called a *probing procedure* or *method*. Compatibility between an extension statement and a probing procedure or method will be defined shortly. The keyword *probing* introduces a probing clause. The syntax of a probing clause is

probing (eC:: eM: eL) pF: pL

A probing clause is not a stand-alone statement. How to use a probing clause to declare the procedure or method will be described shortly. The above clause consists of five optional parts eC::, eM:, eL, pF: and pL, which define parameters eC, eM, eL, pF and pL respectively. We use symbols "::" and ":" to separate the parameters. The specific information carried with the above probing clause depends on the parameters eC, eM, eL, pF and pL, which are interpreted as follows.

- eC is a module or class name if part eC:: is declared in the probing clause. The name may or may not be the same as the module or class that contains the probing clause. The parameter eC is called the *extension class* specified by the probing clause.

- eM is the name of a procedure or method if part eM: is declared. The procedure or method may or may not include the probing clause. The parameter eM is called the *extension method* specified by the probing clause.

- eL is a nonnegative integer such as 0, 3, 26, ... if part eL is declared. We call it the *extension label* specified by the probing clause.

- pF is boolean variable declared in the same module or class that includes the probing clause if the part pF: is declared in the probing clause. Parameter pF is called the *probing condition* of the procedure or method that contains the probing clause.

- pL is a nonnegative integer such as 0, 3, 51, ... if part pL is declared. It labels the probing procedure or method that includes the probing clause. The parameter pL is called the *probing label* or simply the *label* of the procedure or method that contains the probing clause.

All the above five parameters are optional. There are 32 (2^5) kinds of probing clauses which include a probing clause without any parameter.

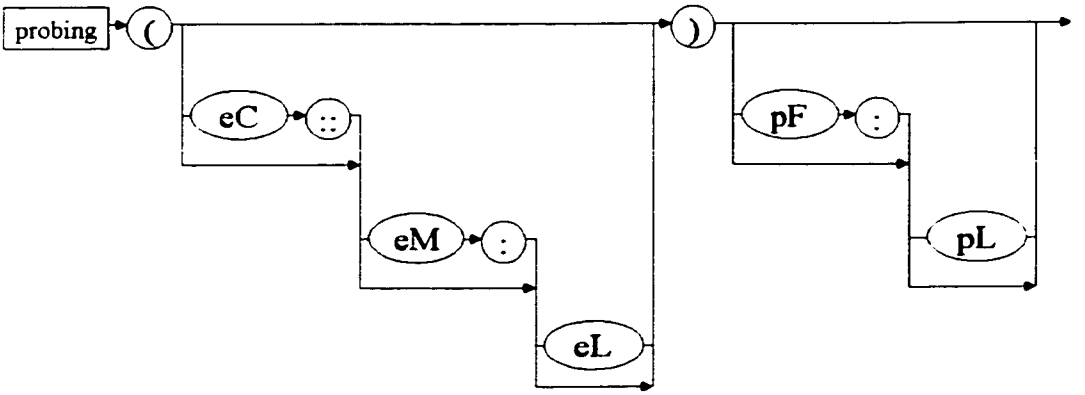

Figure 3.3 Syntax of Probing Clause

The grammar of the probing clause is as follows (see page 25, 26 and 27 for G_1, G_2, G_3 and G_4):

```
G1: Grammar of identifier.
G2: Grammar of number.
G3: Grammar of expression.
G4: Grammar of boolean expression.

Grammar of probing:
G5 = (VN, VT, S, P)
Where VN = {<probing-clause>,<probing>,
           <extension_ps>,<probing_ps>
           <identifier1>,<identifier2>,<expreesion1>,
           <expression2>,<boolean_expression>}
      VT = {probing,"(","::",":",")"}
      S  = <probing-clause>
      P  = {1 <probing-clause>::=<probing>
            2 <probing>::=probing"("<extension_ps>")"
    <probing_ps>
            3 <extension_ps>::=
    <identifier1>"::"|<identifier2>":"|<expression1>
            4 <probing_ps>::=
    <boolean_expression>":"|<expression2>
            5 <identifier1>::=extension_class_name
            6 <identifier2>::=extension_method_name
            7 <expression1>::=extension_label_expression
            8 <expression2>::=probing_label_expression
```

A probing clause is used between a procedure or method declaration and the body of the procedure or method. A normal procedure or method definition

pM (*parameter-list*) { ... }

in C/C++ or Java uses a right parenthesis ")" to terminate its parameter list and uses a left curly brace "{" to start its body. A probing clause is inserted between the right parenthesis ")" and the left curly brace "{". A procedure or method that includes a probing clause is called a *probing method.* For simplicity, we will restrict

each probing procedure or method to have only one probing clause. A probing procedure or method has no parameters. It is not difficult to extend the syntax, semantics and processing techniques for multiple probing clauses in a method. It may require some elaboration to extend the techniques for procedures or methods with parameters.

The following example uses several method definitions to illustrate the syntax of a probing clause. The method bodies are omitted. In the example, we mention only one class name, *Returning_item*, as an extension class parameter in a probing clause.

Example 3.2.2 Method definitions for overload method *item_is_stuck* are:

```
void item_is_stuck()probing(){...};
void item_is_stuck()probing(Returning_item::){...};
void item_is_stuck()probing(Deposit_item_receiver:4)
     {...};
void item_is_stuck()probing(Returning_item::
     Deposit_item_receiver:1)
     item_stuck_on:5{...};
```

Each of the above includes a probing clause. The first probing method does not specify any optional parameters. It represents an extension use case that can be inserted at any position where an extension statement exists in the same program. The second provides only an extension class name, *Returning_item*. Note that class *Returning_item* may or may not include the probing method. The third declares an extension method name, *Deposit_item_receiver*, and an extension label 4. The last probing method contains all the optional parameters. The extension class of the probing method is *Returning_item*. The extension method is *Deposit_item_receiver*. The extension label is 1. The probing condition is *item_stuck_on*. The probing label is 5.

In summary, the syntax of constructs using keywords *extension* and *probing* is as follows.

- Keyword *extension* is used to start a statement. The keyword is followed by a list of parameters within a pair of parentheses. The parameter list is followed by two optional parameters for the extension condition and extension label.

- Keyword *probing* is used in a method definition. The keyword is followed by a parameter list of the method. It introduces a list of optional parameters within a pair of parentheses. The parameter list is followed by two optional parameters for the probing condition and label.

3.3 Semantics of Extension Constructs

The possibility of using one use case to extend another use case is described by means of an extension association between the two use cases. This is described by a probe position in the interaction diagram. A probe position indicates a position in the use case to be extended and is often accompanied by a condition which indicates under what circumstances the extension should take place. The probe thus belongs to the extension use case and not to the one to be extended. This is to avoid changing the original use case when new extensions are added.

The semantics of a probe is as follows. The original use case runs, obeying its description. When the use case reaches the probe position, it checks whether the extension condition is true. If it is, the use case now starts to follow the extension use case's description. When it reaches the end of that description, it returns to its original description and continues where it left off.

3.3.1 Semantics of Extension Statement

Operational semantics describes an abstraction of how the program is executed on a machine. It ignores the details and is independent of machine architectures and implementation details. The operational semantics of extension statement is as follows:

```
S ::= extension(pC:: pM: pL )eF: eL ; |
      pC:=pC'|pM:=pM'|pL:=pL'|eF:=eF'|eL:=eL'

<extension(pC:: pM: pL)eF: eL ;>
          -> <extension(pC'::pM':pL')eF':eL';>
<pC:=pC'> -> <pC'=NULL>|<pC'=pC>|<pC'="NULL">
<pM:=pM'> -> <pM'=NULL>|<pM'=pM>|<pM'="NULL">
<pL:=pL'> -> <pL'=NULL>|<pL'=pL>|<pL'="-1"  >
<eF:=eF'> -> <eF'=NULL>|<eF'=eF>|<eF'="TRUE">
<eL:=eL'> -> <eL'=NULL>|<eL'=eL>|<eL'="-1"  >
```

An extension statement can appear anywhere a normal statement may appear in a program. Without loss of generality, we assume that each statement must be included in a procedure or method, and each procedure or method must be included in a module or class. Thus, for an extension statement

class eC { ...

eM() { ...

extension (pC:: pM: pL) eF: eL ; ... }

};

there are seven pieces of information. The information is described as follows.

- eC is the name of the class or module that includes the extension statement.
- eM is the name of the procedure or method that includes the extension statement.
- eL is the optional extension label of the extension statement. It is a nonnegative integer if the label is defined in the extension statement. Otherwise, we regard the label as -1.
- pC is the optional probing module or class name. It is an identifier if the parameter pC is declared in the extension statement. Otherwise, we regard the value of pC as *null*, which is a reserved word in C/C++ and Java to represent nothing.
- pM is the optional probing procedure or method name. It is an identifier if the parameter pM is declared in the extension statement. Otherwise, we regard the value of pM as *null*.
- pL is the optional probing label of the extension statement. It is a nonnegative integer if the parameter is declared in the extension statement. Otherwise, we regard the probing label as -1.
- eF is the extension condition of the extension statement. It is an identifier if the parameter eF is declared in the extension statement. Otherwise, we regard the value of eF as *true*, which is one of the two values of boolean variable.

Note that the parameters eL, pC, pM, pL and eF have default value -1, *null*, *null*, -1 and *true*, respectively, in an extension statement. The parameters eC and eM are uniquely determined by the location of the extension statement in a program.

3.3.2 Semantics of Probing Clause

The operational semantics of probing clause is as follows:

```
S ::= probing(eC:: eM: eL )pF: pL  |
      eC:=eC'|eM:=eM'|eL:=eL'|pF:=pF'|pL:=pL'

<probing(eC:: eM: eL)pF: pL >
          -> <probing(eC'::eM':eL')pF':pL' >
<eC:=eC'> -> <eC'=NULL>|<eC'=eC>|<eC'="NULL">
<eM:=eM'> -> <eM'=NULL>|<eM'=eM>|<eM'="NULL">
<eL:=eL'> -> <eL'=NULL>|<eL'=eL>|<eL'="-1"  >
<pF:=pF'> -> <pF'=NULL>|<pF'=pF>|<pF'="TRUE">
<pL:=pL'> -> <pL'=NULL>|<pL'=pL>|<pL'="-1"  >
```

A probing clause can appear in a procedure or method definition after the parameter list of the procedure or method. For a probing procedure or method definition

class pC { ...

pM( ) probing (eC :: ***eM*** : ***eL) pF*** : ***pL*** { ... } ;

... }

included in a module or class pC, there are seven pieces of information. The information is described as follows.

- pC is the name of the module or class that includes the probing clause.

- pM is the name of the procedure or method that includes the probing clause.

- pL is the optional probing label of the probing clause. It is a nonnegative integer if the label is defined in the probing clause. Otherwise, we regard the label as -1.

- eC is the name of a module or class that may or may not include the probing method. It is an identifier if the parameter eC is declared in the probing clause. Otherwise, we regard the value of eC as *null*.

- eM is the name of a procedure or method representing an extended use case. It is an identifier if the parameter eM is declared in the probing clause. Otherwise, we regard the value of eM as *null*. The extended procedure or method may or may not be the same as the probing procedure or method.

- eL is the optional extension label of the probing method. It is a nonnegative integer if the label is declared in the probing clause. Otherwise, we regard the label as -1.

- pF is the probing condition of the probing procedure or method. It is an identifier if the parameter pF is defined in the probing clause. Otherwise, we regard the value of pF as *true*.

The parameters pL, eC, eM, eL and pF in a probing clause have default value –1, *null*, *null*, -1 and *true*, respectively. Parameters pC and pM are uniquely determined by the location of the probing clause in a program.

We now describe an optional semantics for the extension statement and probing clause. Let us assume a program, which consists of modules or classes. Extension statements and probing methods may appear in any of the modules or classes. An

extension statement is used in a procedure or method body to indicate a location where probing methods may be applied. Note that we have added only the constructs of an extension statement and probing clause into a programming language. We have not changed any other constructs of the language. We assume that the program can be compiled and executed by temporarily ignoring the extension statements and probing clauses from the program. When we encounter an extension statement at runtime, we will describe what should happen for the extension statement. Thus, we can describe the operational semantics of the extension statement and probing clause.

In the following discussion, we restrict ourselves to an object oriented programming language. The discussion can be easily adapted for a procedural language, like C. We assume probing methods are instance methods. It is not difficult to extend the discussion to class (static) methods.

3.3.3 Semantics of Execution

When an extension statement

extension (pC :: pM : pL) eF : eL ;

is encountered during the execution, the extension statement is said to be *enabled* if the value of extension condition eF is *true*. We assume that the statement is included in method eM of class eC. The extension condition eF must be a boolean variable defined in class eC or it has the default value *true*. If the above extension statement is enabled, each probing method

class ppC { ...

ppM( ) probing (peC :: peM : peL) ppF : ppL { ... } ;

... }

is executed automatically if it is enabled and *compatible* with the extension statement in the same program. The probing method ppM will be executed for a newly created object of class ppC that contains the probing method ppM. The above probing method is said to be *compatible* with the above extension statement if

- pC is *null* or pC is equal to ppC;
- pM is *null* or pM is equal to ppM;
- pL is –1, ppL is –1 or pL is equal to ppL;
- peC is *null* or eC is equal to peC;
- peM is *null* or eM is equal to peM; and
- peL is –1, eL is –1 or eL is equal to peL.

The semantics of the extension statement and probing clause implies that when an enabled extension statement is encountered at runtime, multiple probing methods may run automatically. The number of running probing methods depends on the parameters of the extension statement and the parameters of probing methods in the program. It is a non-deterministic process to decide which of the enabled and compatible methods to run first.

We use the following example to illustrate the extension mechanism automated by the constructs of the extension statement and probing clause.

Example 3.3.3 The following C++ program consists of three classes named *ClassA*, *ClassB* and *ClassC*. The main method starts the execution of the C++ program. The main method contains a couple of extension statements. *ClassA* defines three methods — *am1*, *am2* and *am3*. These three methods are probing methods with different probing parameters. *ClassB* defines two methods, which are not probing methods. *ClassC* defines two methods – *cm1* and *cm2*. The first method is a probing method. The second is not. The following code shows that a probing method can be used to extend an extension statement as well as being invoked as a normal method.

```
void main()
{
  boolean ebma;
  boolean ebmb;

  ebma=FALSE;
  ebmb=TRUE;

  ClassA ca;
  ClassB cb;
  ClassC cc;
  extension(ClassC::)ebma:;
  extension(0)ebmb:;

  cb.bm2();
  cc.cm1();

  cout<<"done! \n\n";
};


class ClassA {
public:
  boolean eba2;
  boolean pba2;
  boolean pba3;
public:
```

```
  ClassA () { eba2=TRUE;
              pba2=FALSE;
              pba3=TRUE;
            };
  void am1( ) probing(){
                          cout<<"am1 done! \n";
                        };

  void am2( ) probing()pba2:;
  void am3( ) probing(main:)1;

};

void ClassA::am2( )
{
  cout<<"am2 done! \n";
  extension(ClassB::);
};

void ClassA::am3( )
{
  cout<<"am3 done! \n";
};

class ClassB {
public:
  boolean ebb1;
public:
  ClassB() {ebb1=TRUE;};
  void bm1( );
  void bm2( );
};

void ClassB::bm1( )
{
  cout<<"bm1 executed.\n";
};

void ClassB::bm2( )
{
  cout<<"bm2 executed.\n";
  extension(1)ebb1:0;
```

```
};

class ClassC {
public:
  boolean pbc1;
  boolean ebc2;
public:
  ClassC() {pbc1=TRUE; ebc2=TRUE;};
  void cm1( )probing(main:)pbc1:1{
                              cout<<"cm1 executed.\n";
                                    };

  void cm2( );
};

void ClassC::cm2( )
{
  cout<<"cm2 executed.\n";
  extension()ebc2:1;
};
```

After the main method is started, statement

```
  extension(ClassC::)ebma:;
```

is executed, the extension condition *ebma* is *true* and the extension statement is enabled. Thus, all the probing methods that are enabled and compatible with the extension statement will be executed automatically. In fact, the only probing method that is compatible with the extension statement is *cm1* in *ClassC*. After creating an instance of *ClassC*, variable *pbc1* in the instance has value *true*. The method *cm1* is enabled and invoked for the object. Probing method *cm1* in *ClassC* may implement an extension use case that executes automatically when a compatible statement is enabled during execution.

Note that we need to check the enabledness of all probing methods for an enabled extension statement. Therefore, a new object is created, referred to as

local variable *x*, for each of the probing methods. Only the enabled probing methods that are compatible with the extension statement will run.

The output of the above program is as follows. The comments indicate which statement produces the output.

```
//statement cb.bm2();
am1 done!
//statement cb.bm2();
bm2 executed.
//statement cc.cm1();
am1 done!
//statement cc.cm1();
cm1 executed.
//statement cout<<"done! \n\n";
done!
```

3.3.4 Program Translation Technique

Based on the semantics of extension statements and probing clauses, we now describe a technique to translate classes that contain extension statements and probing methods into classes with no extension statement and no probing clause. The functionality specified with the extension statements and probing methods will be fulfilled by the resulting classes.

Neither of the identifier *extension* and *probing* is a keyword in programming languages. The identifiers must be converted to legal constructs in a programming language before any source code that contains them can be compiled and executed. The technique translates extension statements into regular statements and eliminates probing clauses from probing methods. The translation follows the operational semantics of the extension constructs.

The translation simply uses probing method invocation to replace extension statements. Each probing method invocation is preceded by a probing condition test. If the test fails, the method invocation will not be performed. An object is created to perform the probing condition test and the method invocation. Assume an extension statement

class eC { ...

eM( ) { ...

extension (pC :: ***pM*** : ***pL) eF*** : ***eL*** ; ... }

... }

in a method eM of class eC. This statement is replaced with a block of statements, one for each probing method,

class ppC { ...

ppM( ) probing (peC :: ***peM*** : ***peL) ppF*** : ***ppL*** { ... };

... }

Assume the above method appears in class ppC and ppF is a boolean instance variable defined in class ppC. The above method is represented with statement

```
{
  ppC x;
  if ( x.ppF &&

    ( (strcmp(ppC,pC)==0)||(strcmp(pC,NULL)==0) )&&
    ( (strcmp(ppM,pM)==0)||(strcmp(pM,NULL)==0) )&&
```

```
   ( (strcmp(pL,ppL)==0)||(strcmp(pL,"-1")==0)
||(strcmp(ppL,"-1")==0) )&&
   ( (strcmp(peC,eC)==0)||(strcmp(peC,NULL)==0) )&&
   ( (strcmp(peM,eM)==0)||(strcmp(peM,NULL)==0) )&&
   ( (strcmp(peL,eL)==0)||(strcmp(peL,"-1")==0)
||(strcmp(eL,"-1")==0) )
     )
  x.ppm();
}
```

in the block for the extension statement. The above statement is simplified when the probing condition *ppF* is known *true* or it is not in the probing clause. In this case, we simply delete the conjunct part *x.ppF* from the above statement. The extension replacement technique is illustrated with the following example.

Example 3.3.4 The following is a C++ program, which consists of two classes. The extension statement in the *main* method and in method *bm1* in *ClassB* specifies no parameters. The probing method *am1* in *ClassA* and *bm1* in *ClassB* define no parameter.

```
void main()
{
  extension( );
}
class ClassA {

public:
  void am1( ) probing(){
                          cout<<"am1 done! \n";
                        };
}

class ClassB {
public:
  void bm1( ) probing( ){
                             cout<<"bm1 is done! \n";
```

```
                    extension( );
                }
}
```

The above compilation unit can be translated to the following equivalent compilation unit, which contains no extension statement or probing clause. For simplicity, we omit the logical expressions in the translated classes. For instance, the extension statement in the *main* method is replaced with statements

```
{
  ClassA x;
  x.am1( );
}
{
  ClassB x;
  x.bm1( );
}
```

rather than the lengthy equivalent statements

```
{
  ClassA x;
  if ( TRUE &&
  ( (strcmp(NULL,NULL)==0)||
    (strcmp("ClassA",NULL)==0)   )&&
  ( (strcmp(NULL,NULL)==0)||
    (strcmp("am1",NULL)==0)      )&&
  ( (strcmp("-1","-1")==0)||
    (strcmp("-1","-1")==0)||
    (strcmp("-1","-1")==0)       )&&
  ( (strcmp(NULL,NULL)==0)||
    (strcmp(NULL,NULL)==0)  )&&
  ( (strcmp(NULL,NULL)==0)||
    (strcmp("main",NULL)==0)     )&&
  ( (strcmp("-1","-1")==0)||
    (strcmp("-1","-1")==0)||
```

```
    (strcmp("-1","-1")==0)       )
  )   //if condition
  x.am1( );
}

{
  ClassB x;
  if ( TRUE &&
  ( (strcmp(NULL,NULL)==0)||
    (strcmp("ClassB",NULL)==0)  )&&
  ( (strcmp(NULL,NULL)==0)||
    (strcmp("bm1",NULL)==0)     )&&
  ( (strcmp("-1","-1")==0)||
    (strcmp("-1","-1")==0)||
    (strcmp("-1","-1")==0)      )&&
  ( (strcmp(NULL,NULL)==0)||
    (strcmp(NULL,NULL)==0)      )&&
  ( (strcmp(NULL,NULL)==0)||
    (strcmp("main",NULL)==0)    )&&
  ( (strcmp("-1","-1")==0)||
    (strcmp("-1","-1")==0)||
    (strcmp("-1","-1")==0)      )
  )   //if condition
  x.bm1( );
}
```

The result of the translated program is as follows.

```
void main()
{            //extension statement be replaced
  {
    ClassA x;
    x.am1( );
  }
  {
    ClassB x;
    x.bm1( );
  }
}
```

```
class ClassA {

public:
  void am1( ) {  //probing clause be deleted
                 cout<<"am1 done! \n";
               };
}

class ClassB {
public:
               //extension statement be replaced
  void bm1( )  {  //probing clause be deleted
                  cout<<"bm1 is done! \n";
                  {
                    ClassA x;
                    x.am1( );
                  }
                  {
                    ClassB x;
                    x.bm1( );
                  }
               }
}
```

In the above translation, a block of probing method invocations is used for each probing method, to replace each extension statement.

The extension statement replacement technique, shown in Example 3.3.4, closely resembles the operational semantics of the extension and probing constructs. However, it has a deficiency. If a program has m extension statements and n probing methods, the number of method invocation statements in the translated program will be $m \times n$. This implies that if the size of a program is $O(n)$, the size of the resulting program is $O(n^2)$. The squared size of the resulting program implies longer compilation time and larger storage requirements.

3.4 Extension Construct Invocation

We now describe a technique for handling extension statements and probing clauses. The technique, called Extension construct invocation, depends on a general feature of object oriented programming languages. When an object of a class is created, a constructor of the class in invoked. The constructor can perform various operations except returning a value. Constructor is a term of C++ or Java. We will use C++ to describe the extension processing technique. It can be easily adapted for Java and other object oriented languages.

The construct invocation technique is described in Figure 3.4.1. Figure 3.4.1(a) indicates that class *Returning item* contains an extension statement. Two probing methods, named *Item is stuck* and *Add paper*, respectively, should extend the extension statement.

The construct invocation technique is illustrated with Figure 3.4.1(b). The technique constructs an auxiliary class, named *Extension*, to hold information on probing clauses. An extension statement is not replaced by probing method invocations. It will be replaced with the creation expression of an object in class *Extension*. The Extension construct invocation uses parameters to encode the parameters of the extension statement. Thus, each extension statement in the original program is mapped to an object creation statement in the modified program. As indicated in Figure 3.4.1(b), the constructor of class *Extension* is responsible for invoking probing methods. In fact, the constructor also tests probing conditions and checks whether a probing method is compatible with the extension statement.

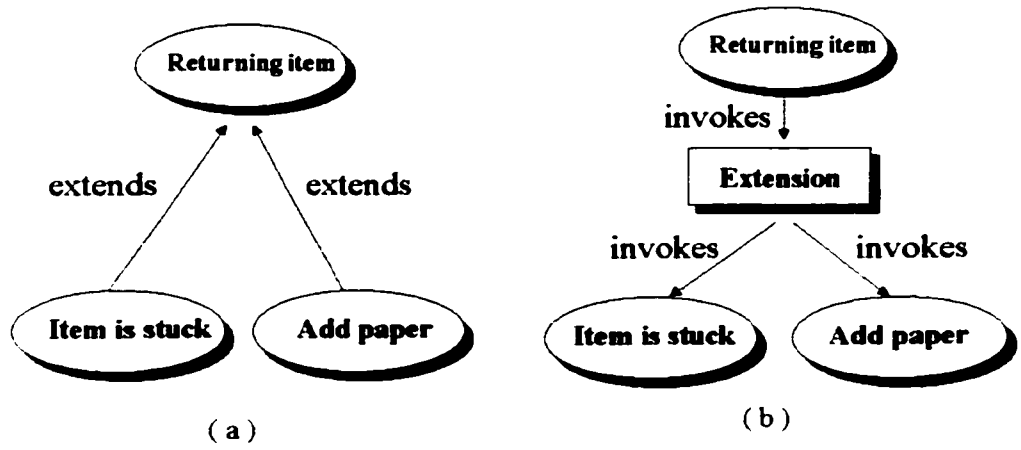

Figure 3.4.1 Extension Class

We use the following example to show the Extension construct invocation technique for extension handling. The example uses the same classes as displayed in Example 3.2.4, each of the two extension statements in the class will trigger both probing methods.

Example 3.4.1 (continuation of Example 3.3.4) The following two classes are the same as the classes in Example 3.3.4. As shown in Example 3.3.4, each of the two extension statements in the classes will trigger both probing methods.

```
void main()
{
  extension( );
}

class ClassA {
```

```
public:
  void am1( ) probing( ){
                              cout<<"am1 done! \n";
                            };
}

class ClassB {
public:
  void bm1( ) probing( ){
                              cout<<"bm1 is done! \n";
                              extension( );
                            }
}
```

The Extension construct invocation technique translates the above 2 classes into the following 3 classes. The extra class, class *Extension*, encodes the probing clauses and invokes the probing methods in its constructor. The parameters of the constructor represent the parameters of an extension statement.

```
void main()
{             //extension statement be replaced
  Extension x(NULL,"main","-1",NULL,NULL,"-1");
}

class ClassA {

public:
  void am1( ) {  //probing clause be deleted
                 cout<<"am1 done! \n";
               };
}

class ClassB {
public:
                  //extension statement be replaced
  void bm1( )  {  //probing clause be deleted
```

```
                    cout<<"bm1 is done! \n";
                    {
  Extension x("ClassB","bm1","-1",NULL,NULL,"-1");
                    }
                }
}

class Extension {
public:
  Extension(char * eC, char * eM, char * eL,
            char * pL, char * pC, char * pM)
  {
  {
    ClassA x;
    if(
        ( (strcmp(NULL,NULL)==0)||
          (strcmp(eC,NULL)==0)       )&&
        ( (strcmp(NULL,NULL)==0)||
          (strcmp(eM,NULL)==0)       )&&
        ( (strcmp("-1","-1")==0)||
          (strcmp(eL,"-1")==0)||
          (strcmp(eL,"-1")==0)       )&&
        ( (strcmp(pC,NULL)==0)||
          (strcmp("ClassA",pC)==0)   )&&
        ( (strcmp(pM,NULL)==0)||
          (strcmp("am1",pM)==0)      )&&
        ( (strcmp("-1","-1")==0)||
          (strcmp(pL,"-1")==0)||
          (strcmp(pL,"-1")==0)       )
      )
      x.am1( );
  }

  {
    ClassB x;
    if(
        ( (strcmp(NULL,NULL)==0)||
          (strcmp(eC,NULL)==0)       )&&
        ( (strcmp(NULL,NULL)==0)||
          (strcmp(eM,NULL)==0)       )&&
        ( (strcmp("-1","-1")==0)||
          (strcmp(eL,"-1")==0)||
```

```
            (strcmp(eL,"-1")==0)        )&&
          ( (strcmp(pC,NULL)==0)||
            (strcmp("ClassB",pC)==0)   )&&
          ( (strcmp(pM,NULL)==0)||
            (strcmp("bm1",pM)==0)       )&&
          ( (strcmp("-1","-1")==0)||
            (strcmp(pL,"-1")==0)||
            (strcmp(pL,"-1")==0)        )
        )
        x.bm1( );
    }
    }
}
```

For a probing method such as *am1* in class *ClassA*, the probing clause is removed from the method. Thus, the method becomes a normal C++ method. The probing clause is translated to a block of statements in the constructor of class *Extension*. The block tests whether the extension statement, for which the constructor is invoked, is compatible with the probing method *am1*. If the extension statement is compatible with *am1*, method *am1* is invoked for a newly created object *x* of class *ClassA*.

A general description of the Extension construct invocation technique is now given. The description consists of two parts, one is for translating extension statements to normal statements, the other is for constructing class *Extension*. Class *Extension* builds constructors from probing clauses.

We now describe how to convert extension statements to normal statements, which may create objects in class *Extension* in order to invoke the constructor of class *Extension*. For each extension statement

extension (pC :: pM : pL) eF : eL ;

in a method eM of class eC, the extension statement is replaced with a block

```
{
  if (eF)
    Extension x(eC1,eM1,eL1,eL1,pC1,pM1,pL1);
}
```

The arguments of the Extension construct invocation encode the parameters of the extension statement. The arguments are generated as follows.

- eC1 is the string "eC".
- eM1 is the string "eM".
- eL1 is equal to the extension label of the extension statement if the statement defines an extension label. Otherwise, it is -1.
- eF1 is equal to eF if eF is declared. Otherwise eF1 takes value *true* and eF1 may be omitted from the block.
- pC1 is string "pC" if parameter pC is declared in the extension statement. Otherwise, pC1 has value *null*. For example, if extension parameter pC is *ClassC*, pC1 is equal to string "*ClassC*".
- pM1 is string "pM" if parameter pM is declared. Otherwise, the value of pM1 is *null*. For example, pM1 may take string "*cm1*", which encodes method name *cm1*.
- pL1 is equal to the probing label pL of the extension statement if a nonnegative integer pL is declared. Otherwise, pL1 is -1.

Comparing with the extension replacement technique, shown in Section 3.3 *Semantics of Extension Constructs*, the Extension construct invocation technique generates concise translation of extension statements. Each statement is converted to one statement in the resulting program.

We now describe how to represent probing methods in the constructor of class *Extension*. Each probing method is represented with a block in the constructor. The block encodes the probing parameters of the method and invokes the method. In the following description of class *Extension*, we show only one block for the probing method

ppM( ) probing (peC:: peM: peL) ppF: ppL { ... }

in the constructor. We assume the method is in class ppC.

```
class Extension {
public:
  Extension(char * eC, char * eM, char * eL,
            char * pC, char * pM, char * pL) {
{
  ppC x;
  if ( x.ppF1 &&

    ( (strcmp(ppC,pC)==0)||(strcmp(pC,NULL)==0) )&&
    ( (strcmp(ppM,pM)==0)||(strcmp(pM,NULL)==0) )&&
    ( (strcmp(pL,ppL)==0)||(strcmp(pL,"-1")==0)
||(strcmp(ppL,"-1")==0) )&&
    ( (strcmp(peC,eC)==0)||(strcmp(peC,NULL)==0) )&&
    ( (strcmp(peM,eM)==0)||(strcmp(peM,NULL)==0) )&&
    ( (strcmp(peL,eL)==0)||(strcmp(peL,"-1")==0)
||(strcmp(eL,"-1")==0) )

      )
  x.ppM( );
}
}
```

}

The block in the *Extension* constructor uses strings and integers to encode the probing method parameters. The constants are described as follows.

- ppF1 is the identifier ppF if ppF is declared in the probing method. Otherwise, x.ppF1 is replaced with *true* and "*true* &&" can be removed.

- ppC1 is string "ppC". For example, for probing method *am1* in class *ClassA*, the class name is converted to string "*ClassA*" for ppC1 in the *Extension* constructor.

- ppM1 is string "ppM". For example, probing method name *am1* is represented with constant ppC1 equal to "*am1*".

- ppL1 is equal to the probing label ppL of the probing method if nonnegative integer parameter ppL is defined. Otherwise, ppL1 is -1.

- peC1 is string "peC" if parameter peC is declared in the probing method. Otherwise, peC1 has value *null*. For example, if parameter peC is class name *ClassA*, peC1 is equal to string "*ClassA*".

- peM1 is string "peM" if parameter peM is declared. Otherwise, peM1 has value *null*. For example, if parameter peM is method name *main*, peM1 is equal to string "*main*".

- peL1 is equal to the nonnegative integer peL of the probing method if the probing method declares as extension label peL. Otherwise, it is -1.

The Extension construct invocation technique translates each probing clause to only one statement, the length of which is comparable with that of the probing clause.

We now describe an implementation of the Extension construct invocation technique. The technique is implemented with a tool called translator, which translates a collection of classes with extension statements and probing methods into classes with no extension statement or probing clause. The tool also constructs class *Extension*. The constructor of class *Extension* encodes the probing clauses with a statement, one for each probing clause. The following procedure formalizes the tool.

Procedure name: translator

Input: A set of classes P, which is the input program P.

Output: A set of classes P′ such that each class C ∈ P is mapped to a class C′ ∈ P′. The set P′ also contains class *Extension*.

Procedure:

Step 1. For each class C ∈ P, replace each extension statement

extension (pC :: pM : pL) eF : eL ;

with statement

```
{
  if (eF)
  Extension x(eC1,eM1,eL1,pC1,pM1,pL1);
}
```

The clause if (eF) can be omited if eF is equal to *true* or not defined by the extension statement.

Step 2. Create class *Extension* with a constructor stub, shown as follows.

```
Class Extension{
Public:
  Extension(char * eC1,char * eM1,char * eL1,
            Char * pC1,char * pM1,char * pL1) {
  }
}
```

For each probing method

ppM( ) probing (peC:: peM: peL) ppF: ppL { ... }

in a class ppC ∈ P, add block

```
{
  ppC x;
  if ( x.ppF1 &&

    ( (strcmp(ppC1,pC)==0)||(strcmp(pC,NULL)==0)   )&&
    ( (strcmp(ppM1,pM)==0)||(strcmp(pM,NULL)==0)   )&&
    ( (strcmp(pL,ppL1)==0)||(strcmp(pL,"-1")==0)
||(strcmp(ppL1,"-1")==0)                            )&&
    ( (strcmp(peC1,eC)==0)||(strcmp(peC1,NULL)==0))&&
    ( (strcmp(peM1,eM)==0)||(strcmp(peM1,NULL)==0))&&
    ( (strcmp(peL1,eL)==0)||(strcmp(peL1,"-1")==0)
||(strcmp(eL,"-1")==0) )
       )
  x.ppM( );
}
```

into the Extension constructor, where constants ppF1, ppC1, ppM1, ppL1, peC1, peM1 and peL1 encode the probing parameters ppF, ppC, ppM, ppL, peC, peM and peL, respectively. Remove the probing clause from the method ppC′ ∈ P′.

Step 3. Set $P' = P \cup \{ \textit{Extension} \}$.

We use the following example to illustrate the Extension construct invocation technique. The example translates a set of classes with extension constructs.

Example 3.4.2 (Continuation of Example 3.3.3) The tool *translator* converts the classes in Example 3.3.3, which have extension and probing constructs, into the following classes. It replaces extension statements with *Extension* constructor invocations. It removes probing clauses from methods. The replaced extension statements are placed in C++ style comments, which start with double slash "//". The removed probing clauses are placed inside C style comments, which start with "/*" and end with "*/".

```
void main()
{
  boolean ebma;
  boolean ebmb;

  ebma=FALSE
  ebmb=TRUE;

  ClassA ca;
  ClassB cb;
  ClassC cc;

  extension(ClassC::)ebma:;
  extension(0)ebmb:;

  cb.bm2();
  cc.cm1();

  cout<<"done! \n\n";
};

class ClassA {
public:
```

```
  boolean eba2;
  boolean pba2;
  boolean pba3;
public:
  ClassA () { eba2=TRUE;
              pba2=FALSE;
              pba3=TRUE;
            };

  void am1( )  /*probing()*/
  {
    cout<<"am1 done! \n";
  };

  void am2( )  /*probing()pba2:*/;
  void am3( )  /*probing(main:)1*/;
};

void ClassA::am2( )
{
  cout<<"am2 done! \n";
  /* extension(ClassB::) */
  { ExtensionProbing x("ClassA", "am2", "-1",
                "No_Extension_Firing", "-1", "ClassB", NULL);
          {
            Extension extension(x);
          };
   } ;
};

void ClassA::am3( )
{
  cout<<"am3 done! \n";
};

class ClassB {
public:
  boolean ebb1;
public:
  ClassB() {ebb1=TRUE;};
  void bm1( );
  void bm2( );
```

```
};

void ClassB::bm1( )
{
  cout<<"bm1 executed.\n";
};

void ClassB::bm2( )
{
  cout<<"bm2 executed.\n";
  /* extension(1)ebb1:0 */
  { ExtensionProbing x("ClassB", "bm2", "0",
                       "ebb1", "1", NULL, NULL);
       if (ebb1)
         {
           Extension extension(x);
         };
   } ;
};

class ClassC {
public:
  boolean pbc1;
  boolean ebc2;
public:
  ClassC() {pbc1=TRUE; ebc2=TRUE;};

  void cm1( ) /*probing(main:)pbc1:1*/
  {
    cout<<"cm1 executed.\n";
  };

  void cm2( );
};

void ClassC::cm2( )
{
  cout<<"cm2 executed.\n";
  /* extension()ebc2:1 */
  { ExtensionProbing x("ClassC", "cm2", "1",
                       "ebc2", "-1", NULL, NULL);
      if (ebc2)
```

```
        {
          Extension extension(x);
        };
    } ;
};
```

In addition to the above modified classes, the tool *translator* also creates class *Extension*, which is shown as follows. The constructor of the class encodes the probing clauses.

```
class Extension{
public:
  Extension(ExtensionProbing x);
}; //class Extension

Extension::Extension (ExtensionProbing x)
{
  {  // probing method am1() in class ClassA
  ExtensionProbing p(NULL, NULL, "-1",
                "No_probing_Firing", "-1", "ClassA", "am1" );
             if(p.isEqual(x)){
               ClassA aClassA
               {
                 aClassA.am1();
               }
             }
  }

  {  // probing method am2() in class ClassA
  ExtensionProbing p(NULL, NULL, "-1",
                   "pba2", "-1", "ClassA", "am2" );
             if(p.isEqual(x)){
                 ClassA aClassA;
                 if (aClassA.pba2)
                 {
                   aClassA.am2();
                 }
             }
  }
```

```
  {  // probing method am3() in class ClassA
  ExtensionProbing p(NULL, "main", "-1",
        "No_probing_Firing", "1", "ClassA", "am3" );
             if(p.isEqual(x)){
                ClassA aClassA;
                {
                  aClassA.am3();
                }
             }
  }

  {  // probing method cm1() in class ClassC
  ExtensionProbing p(NULL, "main", "-1",
           "pbc1", "1", "ClassC", "cm1" );
             if(p.isEqual(x)){
                ClassC aClassC;
                if (aClassC.pbc1)
                {
                  aClassC.cm1();
                }
             }
  }
};

class ExtensionProbing{
public:
  char * EC;
  char * EM;
  char * EL;
  char * BOOLEAN1;
  char * PL;
  char * PC;
  char * PM;
public:
  ExtensionProbing(char * eC, char * eM, char * eL,
                   char * boolean1,
                   char * pL, char * pC, char * pM)
  {
    EC=eC;
    EM=eM;
    EL=eL;
    BOOLEAN1=boolean1;
```

```
    PL=pL;
    PC=pC;
    PM=pM;
  }

boolean isEqual(ExtensionProbing x)
{
  if (
( (strcmp(EC,x.EC)==0)||(strcmp(EC,NULL)==0)    )&&
( (strcmp(EM,x.EM)==0)||(strcmp(EM,NULL)==0)    )&&
( (strcmp(EL,x.EL)==0)||(strcmp(EL,"-1")==0)||
  (strcmp(x.EL,"-1")==0)                        )&&
( (strcmp(PC,x.PC)==0)||(strcmp(x.PC,NULL)==0) )&&
( (strcmp(PM,x.PM)==0)||(strcmp(x.PM,NULL)==0) )&&
( (strcmp(PL,x.PL)==0)||(strcmp(PL,"-1")==0)||
  (strcmp(x.PL,"-1")==0) )
     )
    return TRUE;
  else
    return FALSE;
};

}; //class ExtensionProbing
```

Note that each probing clause is converted to a block, which contains only one statement, in the constructor of class *Extension*. The size of the constructor of class *Extension* is comparable with the number of probing methods in the original classes.

Given a program that may contain extension statements and probing methods, if the size of the program is n, the size of the translation program is $O(n)$, which is linear in the input program size. Thus, the Extension construct invocation technique improves the extension replacement technique shown in Section 3.3 *Semantics of Extension Constructs*.

Example 3.4.3 (Recycling Machine)

```
#include <iostream.h>

enum booleans {FALSE, TRUE};
typedef enum booleans boolean;

boolean Stuck_On = TRUE;
boolean Probing_happen1=TRUE;
    //Probing boolean condition

boolean No_Firing = TRUE;
    //Extension boolean condition


class Item_Return {
   Item_Return();
   Deposit_receiver() {
   //...

   extension(classname::functionname:5)Boolean1:9;
   extension(functionname:)3;
   extension(7);
   extension( );
   //...
   }

};

class Alarmist {
  Alarm  Item_stuck, out_of_paper;
  Alarmist();

  void functionC() probing(funcC:2)aaa:;
  void item_stuck()probing(
    Item_Return::Deposit_receiver:)Stuck_On:9
  {
    cout>>"Alamring: item stuck!";
    extension(classB::funcB:)BooleanB:;

  } //item_stuck
```

```
  //...
  //other functions defination,
  //          no extension statement here

  void funcD() probing(ClasD::3)1;

}; //Alarmist

  void Alarmist::item_stuck()
{
    cout>>"Alamring: item stuck!";

    extension(classA::);
}

void Alarmist::funcC()
{
    //... ...
}


main()
{
    //...
    extension(classB::functionB:)BooleanCondition2:;

}
```

We can translate this program to C++ program without extension statements and probing clauses as following:

```
#include <iostream.h>
enum booleans {FALSE, TRUE};
typedef enum booleans boolean;
boolean Stuck_On = TRUE;
boolean Probing_happen1 = TRUE;
  //Probing boolean condition
boolean No_Firing = TRUE;
  //Extension boolean condition
```

```
class Item_Return {
  Item_Return();
  Deposit_receiver() {
  //...
  /*extension(classname::functionname:5)Boolean1:9*/
  { ExtensionProbing x("Item_Return",
      "Deposit_receiver", "9",
      "Boolean1", "5", "classname", "functionname");
    if (Boolean1)
      Extension extension(x);
  };

  /*extension(functionname:)3*/
  { ExtensionProbing x("Item_Return",
      "Deposit_receiver", "3",
      "No_extension_Firing", "-1",
      NULL, "functionname");
    if (No_extension_Firing)
      Extension extension(x);
  };

  /*extension(7)*/
  { ExtensionProbing x("Item_Return",
      "Deposit_receiver", "-1",
      "No_extension_Firing", "7", NULL, NULL);
    if (No_extension_Firing)
      Extension extension(x);
  };

  /* extension( ) */
  { ExtensionProbing x("Item_Return",
      "Deposit_receiver", "-1",
      "No_extension_Firing", "-1", NULL, NULL);
    if (No_extension_Firing)
      Extension extension(x);
  };
  //...
  }
};
```

```
class Alarmist {
Alarm  Item_stuck, out_of_paper;
Alarmist();
void functionC()/*probing(funcC:2)aaa:*/;

void item_stuck()
/*probing(Item_Return::Deposit_receiver:)Stuck_On:9
*/
{
  cout>>"Alamring: item stuck!";
  /* extension(classB::funcB:)BooleanB: */
  { ExtensionProbing x("Alarmist",
      "item_stuck", "-1",
      "BooleanB", "-1", "classB", "funcB");
    if (BooleanB)
      Extension extension(x);
  } ;
} //item_stuck

//...
// other functions defination,
//      no extension statement here

void funcD()  /*probing(ClasD::3)1*/;
}; //Alarmist

void Alarmist::item_stuck()
{
  cout>>"Alamring: item stuck!";
  /* extension(classA::) */
  { ExtensionProbing x("Alarmist",
      "item_stuck", "-1",
      "No_extension_Firing", "-1", "classA", NULL);
    if (No_extension_Firing)
      Extension extension(x);
  };
}

void Alarmist::funcC()
{
  //... ...
}
```

```
main()
{
  //...
  /* extension(classB::functionB:)BooleanCondition2:
  */
  { ExtensionProbing x(NULL, "main", "-1",
    "BooleanCondition2", "-1",
    "classB", "functionB");
  if (BooleanCondition2)
    Extension extension(x);
  } ;
}



class Extension {

Extension (ExtensionProbing x) {
{
  ExtensionProbing p(NULL, "funcC", "2",
                     "aaa", "-1", "Alarmist", "functionC" );
  if(p.isEqual(x)){
    Alarmist aAlarmist;
    if (aAlarmist.aaa){
      aAlarmist.functionC();
    }
  }
}

{
  ExtensionProbing p("Item_Return", "Deposit_receiver", "-1",
                  "Stuck_On", "9", "Alarmist", "item_stuck" );
  if(p.isEqual(x)){
    Alarmist aAlarmist;
    if (aAlarmist.Stuck_On){
      aAlarmist.item_stuck();
    }
  }
}
```

```
{
  ExtensionProbing p("ClasD", NULL, "3",
        "No_probing_Firing","1","Alarmist","funcD");
  if(p.isEqual(x)){
    Alarmist aAlarmist;
    {
      aAlarmist.funcD();
    }
  }
}
}
}
```

Chapter 4

C++ PROGRAMING IMPLEMENTATION MECHANISMS

4.1 Object Oriented Programming Technique

Object orientation is a technique for developing models. It maps very closely to the way people think about reality. Therefore object oriented models are often easy to understand.

An *object* is a representation of some entity with both behavior and information tied to it. Only the operations that can be performed on the object are seen from the outside of the object. Objects in most object oriented programming languages are abstract data objects. The external interface of an object is the set of operations defined upon it. Most object oriented languages limit external access to an object to invoking the operations defined on the object, and thus support encapsulation.

A class is itself an object and its external interface consists of a set of operations, including operations to create instances.

There are two kinds of relations between the objects - static relations and dynamic relations. *Static relation* is persistent relation, which is aggregated with partition hierarchy. *Dynamic relation* is temporary communication relation communicating by sending event message. An object oriented programming language allows the designer to define new classes of objects. Each object is an instance of one class. An object is represented by a collection of instance variables, as defined by the class. Each class defines a set of named operations that can be performed on the instances of that class. Operations are implemented

by procedures that can access and assign to the instance variables of the target object. Inheritance can be used to define a class in terms of one or more other classes.

Object oriented programming is a programming methodology based on the following key characteristics:

1) Abstraction - providing some form of classes and objects.

Abstraction is the ability to define new types of objects whose behavior is defined abstractly, without reference to implementation details such as the data structure and methods used to represent the objects. Data abstraction prevents an object from being manipulated except by its defined external operations. Data abstraction is a useful form of modular programming. The behavior of an abstract data object is fully defined by a set of abstract operations defined on the object. The user of an object does not need to understand how these operations are implemented or how the object is represented.

2) Inheritance - providing the ability to build new abstractions out of existing ones.

Inheritance complicates the situation by introducing a new category of client for a class. This means that when one client performs objects of the class and operations on objects, there are other clients (class definitions) that inherit from the class. Inheritance is a useful mechanism for reusing code. The decision by the designer to use inheritance for this purpose should be private. Multiple inheritance means that a class can have more than one parent.

3) Run time polymorphism - providing some form of run time binding.

This definition includes all major languages commonly referred to as object oriented such as C++, Eiffel, Smalltalk, Java and many other languages. Classical programming languages without classes, such as C, Fortran and Pascal, are excluded. Languages that lack direct support for inheritance or run time binding, such as Ada are also excluded.

4.2 Object Oriented Programming Language C++

C++ directly supports a variety of programming styles. For this reason, C++ deliberately differs from languages designed to support a single way of writing programs.

C++ is designed to support a range of styles that are useful to object oriented programming:

- Abstraction (Stroustrup 1987) – the ability to represent concepts directly in a program and hide incidental details behind well-defined interfaces. It is the key to every flexible and comprehensible system of any significant size.

- Encapsulation (Snyder 1986) – the ability to provide guarantees that an abstraction is used only according to its specification. Encapsulation is a technique for minimizing interdependencies among separate modules by defining strict external interfaces. A module is encapsulated if clients are restricted by the definition of the programming language to access the module only via its defined external interface. Encapsulation thus assures designers that compatible changes can be made safely. These benefits are especially important for large systems.

- Polymorphism (Granston and Russo 1990) – the ability to provide the same interface to objects with differing implementations.

- Inheritance (Snyder 1986) – the ability to compose new abstractions from existing one. This is one of the most powerful ways of constructing useful abstractions.

- Run time compactness and speed essential for classical systems programming.

These facilities and general properties can be supported in several alternative ways. For example, one programming language may support a facility in its core language while another supports it in a library. Similarly, a facility provided by a run time mechanism in one language may be provided by a compile-time mechanism in another.

4.3 Construct of C++

4.3.1 Classes and Instances

The basic numeric data types in C and C++ are several types of integers and floating-point numbers. These suffice for most of needs, but there are times when the basic types need to be expanded. In C, we would traditionally organize the basic types into a logical structure and write functions to manipulate that structure. In C++, we do the same thing, but we also bind the data description and its algorithms together and set the combination up as a new data type by defining a class. The class consists of a user-defined data structure, member functions, and custom operators.

A class definition only defines the class. It does not set aside any memory to hold any instance of the class. No instance of the class exists until a program declares one. An instance of a class is called an object. The declaration of a class object can contain a list of initializers in parentheses. The declaration of an instance would contain one or several parameters. These parameters will be passed to the class's constructor function.

4.3.2 Members of a Class

A class's members are defined in the class's definition and consist of data members, the constructor and destructor functions and member functions. There are three access specifiers of class members - *private*, *public* and *protected*. Private members can be accessed only by member functions. Public members can be accessed by member functions and by other functions that declare an instance of the class. There are exceptions to these general rules of "friend function". Protected members work the same as private members except when we use class inheritance. Protected members are just like private members until a new class is derived from a base class that has protected members. Protected members are public to derived classes but private to the rest of the program.

The data members of the class may be any valid C++ data types including another class.

Constructor function is the key concept to implement the extension association of the use case driven approach. When an instance of a class comes into scope, a special function called the *constructor* executes if one has been defined. Constructor function always has the same name as the class. It initializes the class object. The run time system provides enough memory to contain the data members of a class when the class comes into scope. The system does not initialize the data members. The class' constructor function must do any initialization that the class requires. The data variable memory goes back to the system when the class goes out of scope.

There must be a constructor function in the class definition with a parameter list of data types that match those of the argument list in the class object declaration. You may define multiple, overloaded constructor functions for a class. Each of

these would have a distinct parameter list. A constructor function returns nothing. You do not declare it a void, but it is void by default.

When a class object goes out of scope, a special function destructor is called. The destructor function name is always the same as the constructor function name with a title character "~" as a prefix. There is only one destructor function for a class. A destructor function takes no parameters and returns nothing.

The member functions of the class are the functions that are declared within the class definition. There are several categories of functions that can appear in a class - regular member functions, friend functions, virtual function, constructor functions and destructor functions. A member function could be private, in which case only other member functions within the same class could call it.

4.3.3 Conversion Class and Conversion Function

Use of C++ data types involves the implicit application of type conversion rules. These implicit conversions come into play in assignments, function arguments, return values, initializers and expressions.

When a function is written that converts any data type to a class, you tell the compiler to use the conversion when the syntax of a statement implies that the conversion should take effect. There are two ways to write a conversion function. The first is to write a special constructor conversion function, the second is to write a member conversion function.

- A construction function that has only one entry in its parameter list is a conversion function. It works in the usual way as a constructor when you declare an object of the class type with a matching initializer argument. We can use the constructor conversion function to convert from a different

type of data to the class in which we define the constructor conversion function.

- A member conversion function can be used to convert from the class in which we define it to a different type of data. A member conversion function uses the C++ *operator* keyword in its declaration.

There are three C++ forms that invoke conversion functions as follows. Conversion functions that convert from one class to another class can be defined.

- Implicit conversion with form:

  Class1 instance = Class2 instance

- Conversion via a cast:

  Class1 instance = (Class1 name) Class2 instance

- Conversion via reverse cast:

  Class1 instance = Class1 name (Class2 instance)

4.3.4 Copy Constructors of C++

A copy constructor is a class member function that executes when you initialize a new object of the class with an existing object of the same class. It is similar to the conversion constructor function. Conversion constructors convert the values in one class object to the format of an object of a different class. Copy constructors initialize the values from an existing object of a class to a new instance object of that same class.

4.4 Syntax of C++

4.4.1 Syntax of C++ Statements

There are only seven kinds of C++ statements as follows. The extension statement will be the eighth kind of statement in C++.

```
1) selection-statement:
     if (expression) statement
     if (expression) statement else statement
     switch (expression) statement
2) asm-statement:
     asm tokens newline
     asm tokens
     asm { tokens; <tokens; > }
3) compound-statement:
     { < declaration-list > < statement-list > }
       declaration-list:
         declaration
         declaration-list declaraction
       statement-list:
           statement
           statement-list
4) expression-statement:
     <expression> ;
5) labeled-statement:
     identifier : statement
     case constant-expression : statement
     default : statement
6) iteration-statement:
     while (expression) statement
     do statement while (expression)
     for(for-init-statement<expression>
;<expression>)  statement
       for-init-stateme
       expression-statement
       declaraction
7) jump-statement:
     goto identifier
     continus;
     break;
     return <expression> ;
```

8) extension-statement (syntax is shown in Section 3.2 *Syntax of Extension Constructs*)

4.4.2 Syntax of C++ Class Declaration

The syntax-directed translation scheme of C++ class declaration is as follows.

```
Class-specifier:
  class-head
  member-list
class-head:
  calss-key <identifier> <base-specifier>
  calss-key calss-name <base-specifier>
member-list:
  member-declaration <member-list>
  access-specifier : <member-list >
member-declaration:
  <decl-specifiers> <member-declarator-list> ;
  function-definition <;>
  qualified-name;
member-declarator-list:
  member-declarator%
  member-declarator-list,
member-declarator
  member-declarator:
  declarator <pure-specifier>
  <identifier> : constant-expression
pure-specifier:
  = 0
base-specifier:
  : base-list
base-list:
  base-specifier
  base-list, base-specifier
base-specifier:
  class-name
  virtual <access-specifier> class-name
  access-specifier <virtual> calss-name
access-specifier: private, public, protected
```

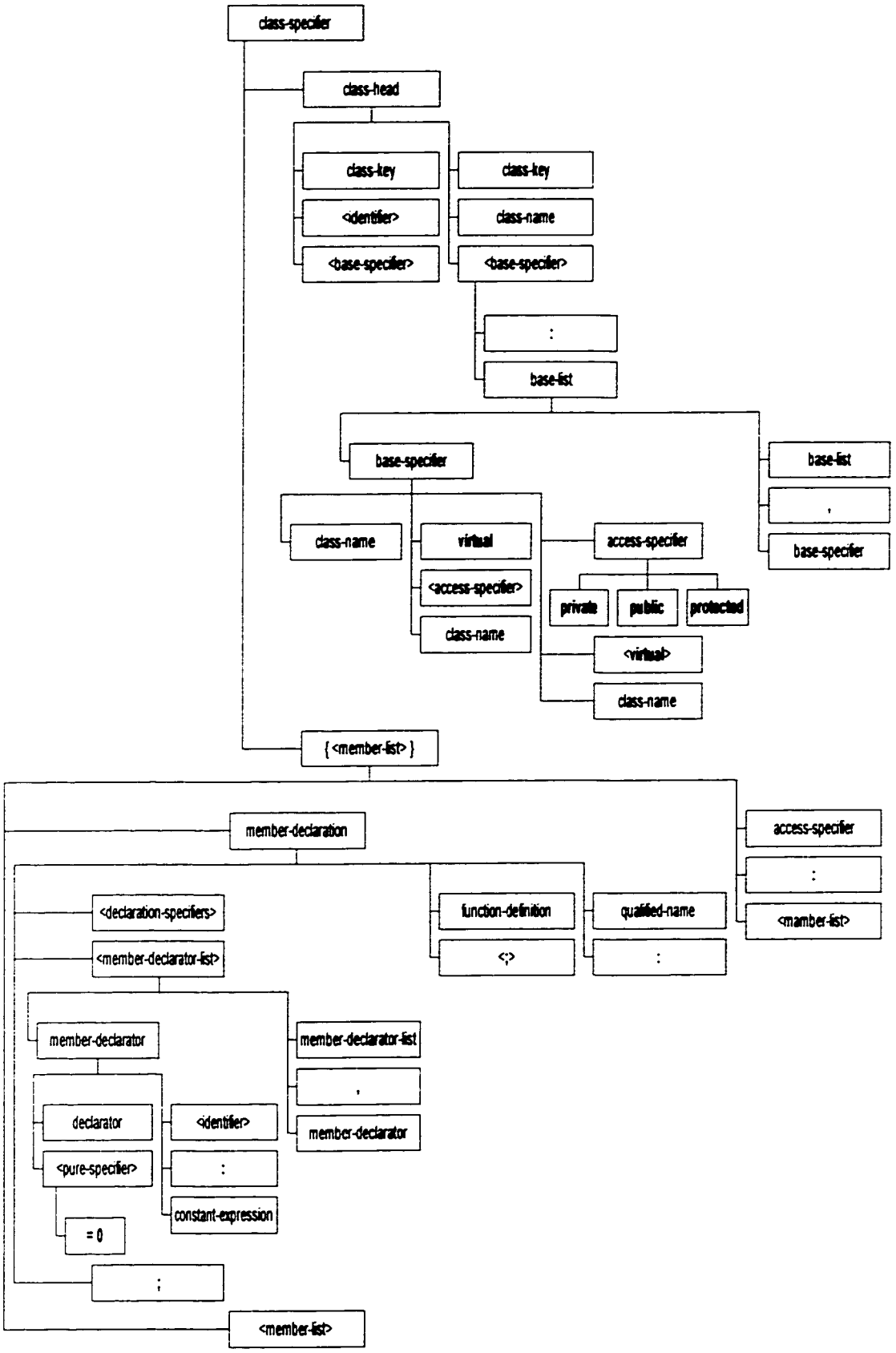

Figure 4.4.2 C++ Class Declaration Grammar

According to the analysis of C++ class declaration grammar, an extension statement can show up only in three situation as follows.

- In the body of *main*( ) function.
- In the body of inline function.
- In the body of outline function with class resolution scope "::".

A probing clause can be show only in the function definition statement.

4.4.3 Hashing Probing Parameters for Extension Processing

4.4.3.1 Hash Table and Probing Parameter Hashing

In general, we assume a vector of length M and a collection of K different possible key values. We furthermore assume no knowledge of relative frequencies of different keys, and therefore take those frequencies to be equal for all keys. The job of a good hash function is to map the keys as equally as possible into the vector elements, so that each element is obtained from roughly K/M different key values. We assume that the time required to find the record in the hash table, which corresponds to a given hash key, is directly related to the number of hash keys.

Even though frequencies are assumed for different keys, we should remain aware of the popularity of collections of names differing only at the start, or perhaps end. We frequently encounter collections of keys such as NAME1, NAME2, ... , or CODEa, CODEb, ... In order to avoid generating identical hash codes for all members of such a group, it is important to define the hash function in such a way that its value is sensitive to all parts of the operand key. It would not suffice to use a hash function based on only the first few, or last few, characters of a key.

The following statement is used to decide the hash key value.

```
for (size_t i = 0; i < length_key; ++i)
{
    n <<= 1;
    n += key[i];
}
key_value = (n % (unsigned long)buckets);
```

The Extension construct invocation technique, introduced in Chapter 3 *Design of Extension Constructs*, has a deficiency. Each probing method is represented with a block statement in the constructor of class *Extension*. The block compares the probing and extension parameters. Based on the comparison, it may invoke the probing method for a newly created object. The blocks are linearly listed inside the constructor. When an extension statement is encountered at runtime, the *Extension* constructor is invoked. All the blocks in the constructor must be examined in sequence. Even if a probing clause explicitly specifies an extension class that is not the class of the extension statement, the probing method parameters must be examined by the constructor. Thus, all the available probing methods are tested in sequence for each extension statement encountered at runtime.

To overcome this deficiency, we present an approach to organize the probing information into hash tables. When the extension constructor is invoked, the extension parameters are used to construct keys, which are used to search for probing methods compatible with the extension statement. Thus, we can find all the compatible probing methods quickly with a hashing key searching algorithm.

As illustrated in Figure 4.4.3.1, the approach uses a hash table, referenced with variable *maintable*. The key of *maintable* is string "eC, eM, eL, pF, pL, pC, pM". The value of *maintable* is "pC,pM". When an extension statement is encountered

at runtime, extension parameters are used to compose keys "eC, eM, eL, pF, pL, pC, pM" to retrieve the probing class with probing method "pC,pM". Each probing condition pF is tested against a newly created object of class pC. If the test result is TRUE, probing method pM is invoked for the created object.

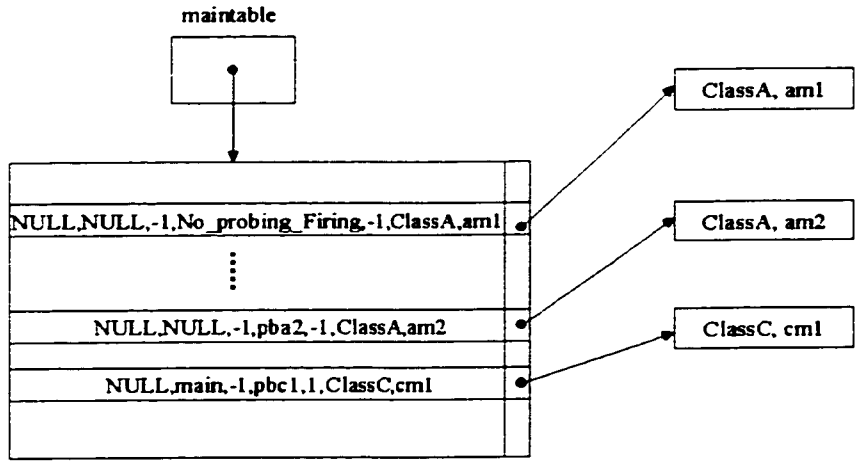

Figure 4.4.3.1 Probing Parameter Hashing

4.4.3.2 Hashing Probing Method Parameters

A framework for the *Extension* class to store probing parameters in the main hash table is now described. A static initialization block in class *Extension* places keys and values into the hash tables. When the class *Extension* is loaded into a C++ runtime system, a hash table is created and assigned to the static (class) variable *maintable*. Then, the static initialization block is executed. In the static initialization block, we have a statement

```
HashTableStrDB maintable(20);

ht.Insert("eC,eM,eL,pF,pC,pM,pL","pC,pM");
```

for each probing method

pM() probing (eC:: eM: eL) pF: pL { ... }

The method *Insert* adds or changes key-value pairs in the *maintable*. The key encodes the probing parameters. The structure of class *Extension* is shown as follows. Only one *Insert* method invocation is displayed in the static initialization block. Others are omitted. The body of constructor is also omitted.

```
Class Extension {
  ...
  //create hash table
  HashTableStrDB maintable(20);


  //insert key-value pair into hash table
  ht.Insert("NULL,main,-1,pbc1,1,ClassC,cm1",
            "ClassC,cm1");

  ...
}
```

4.4.3.3 Retrieving Method Descriptions

We now discuss how to retrieve the descriptions of compatible probing methods from the hash tables when an extension statement

extension (pC:: pM: pL) eF: eL;

is encountered in a method eM of a class eC at runtime. The extension statement will be replaced with block

```
{
  if (eF1)
  Extension x(eC1,eM1,eL1,pC1,pM1,pL1);
}
```

which invokes the extension constructor. The extension parameters eF, eC, eM, eL, pC, pM and pL are mapped arguments eF1, eC1, eM1, eL1, pC1, pM1 and pL1 for the constructor invocation. The mapping follows the rules specified in Chapter 3 *Design of Extension Constructs.*

4.4.3.4 Program Translation with Hash Table

With the above discussion, we can describe the technique of probing parameter hashing for extension processing. Assume a program P that consists of classes, which contain extension statements and probing methods. The technique translates each extension statement into a statement that creates an object by invoking constructor of class *Extension.* It converts each probing clause into a statement in the static initialization block of class *Extension.* The statement invokes method *Insert* to add the probing method to hash tables. Each of the steps of the technique is detailed as follows. The result is a program P′ that contains no extension statement or probing clause. Thus, P′ can be compiled with a compiler such as C++ or Java compiler and the compiled code can be executed. C++ is used in the following description.

Step 1. Create a file extension.cpp that contains the incomplete code of class *Extension.*

```
Class Extension {
  ...
  //create hash table
  HashTableStrDB ht(20); ...
```

Step 2. Each class C in program P is converted to a class C′ in P′. If a method m in class C contains an extension statement

extension (pC :: pM : pL) eF : eL ;

replace the extension statement with statement

```
{
  if (eF1)
  Extension x(eC1,eM1,eL1,pC1,pM1,pL1);
}
```

in class C′, where

- eF1 is *true* if eF: does not appear in the extension statement. Otherwise eF1 is eF;
- eC1 is "C";
- eM1 is "m";
- eL1 is -1 if eL does not appear. Otherwise eL1 is equal to eL;
- pC1 is *null* if pC:: does not appear. Otherwise pC1 is string "pC";
- pM1 is *null* if pM:: does not appear. Otherwise pM1 is string "pM", and
- pL1 is -1 if pL does not appear. Otherwise pL1 is equal to pL.

If class C contains a probing method

pM() probing (eC :: eM : eL) pF : pL { ... }

replace the method with an ordinary method

pM ( ) { ... }

in class C′ by removing the probing clause and appending the statement

```
{
    ht.Insert("eC,eM,eL,pF,pC,pM,pL","pC,pM");
}
```

at the end of file Extension.cpp, where

- pF1 is TRUE if pF: does not appear in the probing clause. Otherwise pF1 is equal to pF;
- eC1 is *null* if eC:: does not appear. Otherwise eC1 is string "eC";
- eM1 is *null* if eM:: does not appear. Otherwise eM1 is string "eM";
- eL1 is -1 if eL does not appear. Otherwise eL1 is equal to eL.
- pC1 is string "C ";
- pM1 is string "m", and
- pL1 is -1 if pL does not appear. Otherwise pL1 is equal to pL.

Step 3. Finish the class *Extension*. We can use the following statements to search hash value via a hash key.

```
//look for entry in the table
String name1("NULL,NULL,-1,pba2,-1,ClassA,am2");
KeyString key1(name1);
```

```
DataBlock db1(ht.LookUp(key1));
```

Step 4. The program P′ consists of classes C′ and class *Extension*. Since P′ contains no extension statement or probing clause, it can be compiled and executed. Method *Lookup*( ) is defined by program hash.h and hash.cpp.

Chapter 5

CONCLUDING REMARKS

The technique Extension Construct Invocation described in this thesis meets all of its goals:

- Efficiency: Using a component from this automatic translation incurs little overhead beyond "hand coded" versions. Real environments and large applications pay no overhead (time or space) for using this technique.

- Ease of Use: This general technique provides a clear organization for the project. The template mechanism provides a convenient way to combine extension statement and probing clause. A person only knows a little about use case and object oriented programming can understand and use it easily.

- Extendibility: The project is extendable through the application of inheritance. It is possible to extend the project with new data structures, new storage managers, or both.

In this study, constructs have been introduced into C++ programming language to support use case extension and support program extension. The keyword *extension* is used to identify a location in a program where probing methods may insert themselves. An extension statement may declare a class name, a probing method name and/or a probing label for restricting probing methods that can be inserted at the location. It may also declare an extension condition and/or an extension label. Probing methods may use the extension label. The keyword *probing* is used to introduce a probing clause for a method. The probing clause

may optionally declare a class name, a method name and an extension label for extension statements that the probing method can extend. It may also define a probing condition and/or a probing label for the probing clause. Extension statements may use the probing label.

An operational semantics has been defined for the extension and probing constructs. The operational semantics can be applied to translate a program that contains extension statements and probing methods into a program with no extension statement or probing clause. However, the size and the time complexity of the resulting program may be squared comparing with that of the original program.

The technique named Extension Construct Invocation is proposed, which converts programs that use extension statements and probing methods to programs with no extension statements or probing method. Furthermore, we propose a method that applies hashing techniques to store the probing method description into hash tables. When an extension statement is encountered at runtime, the compatible probing method description can be retrieved from the hash tables. It is very easy and quick to execute the probing method from the hash table.

The extension and probing constructs are general constructs for programming languages that support module or class, procedure, function or method. Programmers can use these constructs in their code. A tool based on the techniques can be applied to process the extension and probing constructs automatically. Prospects of using the constructs include increased reusability and improved extendibility of software systems.

The implementation of this technique relies on only a couple of simplifying assumptions about the syntax of C++. For example, it assumes no inner classes

in the C++ program. This project can be easily extended for the full version of C++ language. This technique shows that program extension can be made easy if viewed different by from the traditional way of program augmenting and integration.

REFERENCES

Aho, A. V., Sethi, R. and Ullman, J. D., *Compilers – Principles, Techniques, and Tools*, Addison-Wesley Publishing, 1986

Arnold, K. and Gosling, J., *The Java Programming Language*, Addison-Wesley, Reading, Massachusetts, 1996

Backus, J. W., *Transcript of Presentation on the History of Fortran I, II, and III*, Wexelblat, 1981

Booch, G., *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin/Cummings Publishing, Redwood City, California, 1994

Buxton, J. and McDemid, J., *Architectural Design*, Software Engineer's Reference Book, ed. J. McDemid, CRC Press, Inc., 1993

Coleman, D. and Gallimore, R. M., *Software Engineering Using Executable Specifications*, Macmillan Computer Science Series, 1987

Ellis, M.A. and Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley Publishing, Reading, Massachusetts, 1992

Ghezzi, C., Jazayeri, M. and Mandrioli, D., *Fundamentals of Software Engineering*, Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1991

Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Publishing, Reading, Massachusetts, 1983

Goldberg, A. and Robson, D., *Smalltalk-80: The Language Implementation*, Addison-Wesley Publishing, Reading, Massachusetts, 1989

Gosling, J., Joy, B. and Steele, G., *The Java Language Specification*, Addison-Wesley, Reading, Massachusetts, 1996

Granston, E. D. and Russo, V. F., *Signature-Based Polymorphism for C++*, USENIX C++ Conference Proceedings, pp. 65-79, 1990

Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G., *Object-Oriented Software Engineering – A Use Case Driven Approach*, Addison-Wesley Publishing, Reading, Massachusetts, 1992

McDermid, J. and John, P., *Software Development Process Models*, Software Engineer's Reference Book, ed. J. McDermid, CRC Press, Inc., 1993

Parnas, D. L. and Weiss, D. M., *Active design reviews: principles and practices*, Journal of Systems and Software, Vol. 7, No. 4, pp. 259-265, December 1987,

Rumbaugh, I., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W., *Object-Oriented Modeling and Design*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991

Snyder, A., *Encapsulation and Inheritance in Object-Oriented Programming Languages*, SIGPLAN Notices, pp. 38-45, November 1986

Stokes, D. A., *Requirement Analysis*, Software Engineer's Reference Book, ed. J. McDermid, CRC Press, Inc., 1993

Stroustrup, B., *The C++ Programming Language*, Second Edition, Addison-Wesley Publishing, Reading, Massachusetts, 1991

Stroustrup, B., *What is 'Object-Oriented Programming'?*, Proc. 1st European Conference on Object-Oriented Programming, Springer Verlag Lecture Notes in Computer Science, Vol. 276, pp. 51-70, Paris, 1987

Wichmann, B. A., *Implementation*, Software Engineer's Reference Book, ed. J.A. McDermid, CRC Press, Inc., 1993

APPENDIX

Programs list

```
// boolean.h

#ifndef BOOLEAN_H
#define BOOLEAN_H

enum boolean
{
      FALSE=0,
      TRUE=1
};

#endif
```

```
// ep.h

#include <string.h>
#include "boolean.h"

typedef char STRING[250];

#ifndef EP_H
#define EP_H

class ExtensionProbing{
public:
      char * EC;
      char * EM;
      char * EL;
      char * BOOLEAN1;
      char * PL;
      char * PC;
      char * PM;

public:
      ExtensionProbing(char * eC, char * eM, char * eL,
                  char * boolean1,
                  char * pL, char * pC, char * pM)
      {
            EC=eC;
            EM=eM;
            EL=eL;
            BOOLEAN1=boolean1;
            PL=pL;
            PC=pC;
            PM=pM;
      }



      boolean isEqual(ExtensionProbing x)
{
      if (
            ( (strcmp(EC,x.EC)==0)||(strcmp(EC,NULL)==0) )&&
            ( (strcmp(EM,x.EM)==0)||(strcmp(EM,NULL)==0) )&&
( (strcmp(EL,x.EL)==0)||(strcmp(EL,"-1")==0)||(strcmp(x.EL,"-1")==0) )&&
            ( (strcmp(PC,x.PC)==0)||(strcmp(x.PC,NULL)==0) )&&
            ( (strcmp(PM,x.PM)==0)||(strcmp(x.PM,NULL)==0) )&&
( (strcmp(PL,x.PL)==0)||(strcmp(PL,"-1")==0)||(strcmp(x.PL,"-1")==0) )
      )
            return TRUE;
      else
            return FALSE;

};

}; //class ExtensionProbing

#endif
```

```
/*     filename: translator.cpp
  translate cppp file with statements "extension" and "probing"
  to regular c++ and .h files */

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

#include "str.h"
#include "EP.h"

typedef unsigned int word;
typedef unsigned char byte;

STRING line;

FILE *infile_var;
FILE *outfile_var;
FILE *extensionfile_var;

boolean inlineFunction=FALSE;
boolean OneLine=FALSE; /*only one line inline function*/
boolean Probing_declaration_only = FALSE;

class translator : public String {
public:
  translator() {};
  ~translator() {};

  STRING className;
  STRING functionName;

  treat_class(STRING ClassName);
  treat_probing(STRING ClassName, STRING FunctinName);
  treat_function(STRING ClassName, STRING FunctionName);
  translate();
  makefiles();

}; //class translator

main()
{
  translator mt;

  STRING infile0;
  STRING outfile0;
  STRING extensionfile0;

  strcpy(infile0,"try.c3");
  strcpy(outfile0,"try.cpp");
  strcpy(extensionfile0,"exten.cpp");

  infile_var=fopen(infile0,"rt");
  outfile_var=fopen(outfile0,"wt");
  extensionfile_var=fopen(extensionfile0,"wt");

  mt.translate();

  fclose(infile_var);
```

```
  fclose(outfile_var);
  fclose(extensionfile_var);

  strcpy(infile0,"try.cpp");
  infile_var=fopen(infile0,"rt");
  mt.makefiles();

  cout<<"\nThank you, Bye!"<<endl;

} //main()

translator::makefiles()
{
  STRING classname,longstring,mainlong,extenlong;
  STRING outfile0;
  int i,index1,index2;

  strcpy(mainlong,"//filename: main.cpp\n");
  strcat(mainlong,"#include <iostream.h>\n");
  strcat(mainlong,"#include \"boolean.h\"\n");
  strcat(mainlong,"#include \"ep.h\"\n");
  strcat(mainlong,"#include \"exten.h\"\n");

  strcpy(extenlong,"//filename: exten.h\n\n");
  strcat(extenlong,"#include \"ep.h\"\n");

  while (!feof(infile_var))
  {
    fgets(line,80,infile_var);
    if (strlen(line)>0)
    {
      while ( (i=pos_str(line,"class",0))>-1 )
      {
      index1=pos_char(line,' ',i+5);
      index2=pos_char(line,' ',index1+1);
      strcpy(classname,line);
      in_pos_mid(classname,index1+1,index2-1); /*classname*/

      strcat(mainlong,"#include \"");
      strcat(mainlong,classname);
      strcat(mainlong,".h\"\n");

      strcat(extenlong,"#include \"");
      strcat(extenlong,classname);
      strcat(extenlong,".h\"\n");

      strcpy(outfile0,classname);
      strcat(outfile0,".h");
      outfile_var=fopen(outfile0,"wt");

      strcpy(longstring,"//filename: ");
      strcat(longstring,classname);
      strcat(longstring,".h\n\n");
      strcat(longstring,"#include <iostream.h>\n");
      strcat(longstring,"#include \"boolean.h\"\n");
      strcat(longstring,"#include \"ep.h\"\n\n");
      strcat(longstring,"#ifndef ");
      strcat(longstring,classname);
      strcat(longstring,"_H\n");
      strcat(longstring,"#define ");
      strcat(longstring,classname);
```

```
strcat(longstring,"_H\n\n");

fprintf(outfile_var,"%s",longstring);

while( (i=pos_str(line,"};",0))!=0 )
{
  fprintf(outfile_var,"%s",line);
  fgets(line,80,infile_var);
}
fprintf(outfile_var,"%s",line); // "};"

strcpy(longstring,"\n#endif\n");
fprintf(outfile_var,"%s",longstring);

fclose(outfile_var);

fgets(line,80,infile_var);

strcpy(outfile0,classname);
strcat(outfile0,".cpp");
outfile_var=fopen(outfile0,"wt");

strcpy(longstring,"//filename: ");
strcat(longstring,classname);
strcat(longstring,".cpp\n\n");
strcat(longstring,"#include \"");
strcat(longstring,classname);
strcat(longstring,".h\"\n");
strcat(longstring,"#include \"ep.h\"\n");
strcat(longstring,"#include \"exten.h\"\n\n");

fprintf(outfile_var,"%s",longstring);

while( ( (i=pos_str(line,"class",0))==-1
       )
       &&
       ( (i=pos_str(line,"main",0))==-1
       )
     )
{
  fprintf(outfile_var,"%s",line);
  fgets(line,80,infile_var);
}

fclose(outfile_var);

}/* while find classes*/

if ( (i=pos_str(line,"main",0))>-1 )
{
strcpy(outfile0,"main.cpp");
outfile_var=fopen(outfile0,"wt");
fprintf(outfile_var,"%s",mainlong);

while( (i=pos_str(line,"}",0))!=0 )
{
  fprintf(outfile_var,"%s",line);
  fgets(line,80,infile_var);
}
fprintf(outfile_var,"%s",line); // "}"
fclose(outfile_var);
```

```
      strcpy(outfile0,"exten.h");
      outfile_var=fopen(outfile0,"wt");

      strcat(extenlong,"\n#ifndef ");
      strcat(extenlong,"EXTEN_H\n");
      strcat(extenlong,"#define ");
      strcat(extenlong,"EXTEN_H\n\n");
      strcat(extenlong,"class Extension{\npublic:\n");
      strcat(extenlong,"\tExtension(ExtensionProbing x);\n");
      strcat(extenlong,"}; //class Extension\n\n");
      strcat(extenlong,"#endif");

      fprintf(outfile_var,"%s",extenlong);
      fclose(outfile_var);

      } //find "main"

    }// if text exist
  }// while

} //makefiles()

translator::translate()
{
  int i,j,k,index1,index2;
  int leng=0;

  STRING classname, Name0;
  STRING longs;

  strcpy(longs,NULL);
  strcat(longs,"//filename: exten.cpp\n\n");
  strcat(longs,"#include \"exten.h\"\n");
  strcat(longs,"#include \"boolean.h\"\n\n");

  strcat(longs,"Extension::Extension (ExtensionProbing x)\n{\n");
  fprintf(extensionfile_var,"%s",longs);

  strcpy(className,"NULL");
  strcpy(functionName,"NULL");

  while (!feof(infile_var))
  {
    fgets(line,80,infile_var);
    if (strlen(line)>0)
    {
      if ( (i=pos_str(line,"class",0))>-1 )
      {
      index1=pos_char(line,' ',i+5);
      index2=pos_char(line,' ',index1+1);

      strcpy(classname,line);
      in_pos_mid(classname,index1+1,index2-1); /*classname*/

      j=pos_char(line,'{',index2);  /* class begin '{' */
      if (j>-1)
      {
        fprintf(outfile_var,"%s",line);
        treat_class(classname); /* class */
      }
```

```
      }/*if "class" is found*/
      else
      {
      i=pos_str(line,"::",0);
      if (i>-1)
      {
        strcpy(Name0,line);

        leng=strlen(Name0);
        j=pos_char(Name0,':',0);
        in_delete(Name0,j,leng-j); /* "::" */

        k=pos_char(Name0,' ',0);
        while (k>-1)
        {
          in_delete(Name0,0,k+1);    /* delete ' ' */
          k=pos_char(Name0,' ',0);
        }

        strcpy(className,Name0);   /*get class name*/

        strcpy(Name0,line);
        leng=strlen(Name0);
        i=pos_str(line,"::",0);
        in_delete(Name0,0,i+2);

        leng=strlen(Name0);
        j=pos_char(Name0,'(',0);
        in_delete(Name0,j,leng-j);

        strcpy(functionName,Name0);  /*get function name*/

        inlineFunction=FALSE;
        treat_function(className,functionName);
      } /*if "::" is found*/
      else
      {
        i=pos_str(line,"main",0);
        if (i>-1)
        {
          fprintf(outfile_var,"%s",line);
          fgets(line,80,infile_var);

          inlineFunction=FALSE;
          treat_function("NULL","main");
        } /*if "main" is found*/
        else
        {
          fprintf(outfile_var,"%s",line);
        }; /*else "main"*/
      }; /* else "::" */
      }; /* else class */

    }; /*if text exist*/
  }/*while*/

  strcpy(longs,NULL);

  strcat(longs,"};\n");
  fprintf(extensionfile_var,"%s",longs);
} // translate()
```

```
translator::treat_class(STRING ClassName)
{
  int i,j,k,l,m;
  int depth=1;  /*depth of { } */
  int leng;
  STRING functionName, Name0;
  STRING line0;

  depth=1;
  fgets(line,80,infile_var);
  strcpy(line0,line);

  if (strlen(line)>0)
  {
    while (depth != 0)
    {
      i=pos_char(line,'{',0); /*function begin*/
      if (i>-1)               /*function name begin*/
      {
      strcpy(functionName,NULL);

      if ( (m=pos_char(line,'}',0)) > -1 )
        OneLine=TRUE;/* one line inline function {...}; */

      strcpy(Name0,line);

      leng=strlen(line);
      j=pos_char(Name0,'(',0);
      in_delete(Name0,j,leng-j);

      k=pos_char(Name0,'\t',0);
      if ( k>-1 )
        in_delete(Name0,0,1);     /* delete TAB */

      k=pos_char(Name0,' ',0);
      while (k>-1)
      {
        in_delete(Name0,0,k+1);     /* delete ' ' */
        k=pos_char(Name0,' ',0);
      }
      strcpy(functionName,Name0); /*get functionname*/

      // create file named exten.cpp to save class extension
      j=pos_str(line,"probing",0);
      if (j>-1)   /* include "probing" */
      {
        k=pos_char(line,';',0);
        if (k>-1)
        {
          Probing_declaration_only=TRUE;
          treat_probing(ClassName,functionName);
          Probing_declaration_only=FALSE;
        } //function declaration, no body, no extension
        else
        {
          treat_probing(ClassName,functionName);
          treat_function(ClassName,functionName);
        } //in line function, maybe has "extension" statement
      }
      else
```

```
        treat_function(ClassName, functionName);
        // function maybe has "extension" statement

      depth--;

      if (OneLine==TRUE)
        OneLine=FALSE;

      } /*if i, if function begin "{" */
      else
      {
      if ( j=pos_str(line,"probing",0)>-1 )
      {
        if ( k=pos_char(line,';',0)>-1 )
        {
          Probing_declaration_only=TRUE;

          strcpy(functionName,NULL);
          strcpy(Name0,line);

          leng=strlen(line);
          j=pos_char(Name0,'(',0);
          in_delete(Name0,j,leng-j);

          l=pos_char(Name0,'\t',0);
          if ( l>-1 )
            in_delete(Name0,0,1);    /* delete TAB */

          l=pos_char(Name0,' ',0);
          while (l>-1)
          {
            in_delete(Name0,0,l+1);    /* delete ' ' */
            l=pos_char(Name0,' ',0);
          }
          strcpy(functionName,Name0); /*get functionname*/
          treat_probing(ClassName, functionName);
        } //if k
      } //if j
      else
        fprintf(outfile_var,"%s",line);

      fgets(line,80,infile_var);
      strcpy(line0,line);

      if ( (m=pos_char(line,'{',0)) > -1 )
      {
        depth++;        /*inline functions   '{' ...} */
        inlineFunction=TRUE;
      }
      else
      {
        inlineFunction=FALSE;
        if ( (m=pos_char(line,'}',0)) > -1 )
          depth--;        /*end of class    '}' */
      }
      }  /* if i ... else */
    }  /*while not end of class*/

    fprintf(outfile_var,"%s",line);  /* end of class "};" */

  } /*if*/
```

```
}; /*function treat_class */

translator::treat_probing(STRING ClassName, STRING functionName)
{
  STRING eC;
  STRING eM;
  STRING eL;

  STRING probingBoolean;

  //pC: ClassName;
  //pM: functionName;
  STRING pL;

  STRING longstring;
  STRING es; // constructors of Extension

  int i,j,k,l,m;
  int a,b,c;
  int leng;

  STRING line1;
  STRING line2;
  STRING line0;

  boolean eC_exist = FALSE;
  boolean eM_exist = FALSE;
  boolean probingBoolean_exist = FALSE;

  strcpy(eC,"NULL");
  strcpy(eM,"NULL");
  strcpy(eL,"-1");

  strcpy(pL,"-1");
  strcpy(probingBoolean,"No_probing_Firing");

  strcpy(es,"\n\t{\n\tExtensionProbing p(");
  strcpy(longstring,NULL);

  strcpy(line1,line);
  k=pos_char(line1,'(',0);
  in_delete(line1,k,1);
  k=pos_char(line1,')',0);
  in_delete(line1,k,1);
  strcpy(line0,line1);

  k=pos_char(line1,'(',0);
  if (k>-1)
  {
    l=pos_char(line1,')',0);

    if (l!=k+1) //is not just "probing()"
    {
    in_pos_mid(line1,k+1,l-1); //line1 is ( "eC::eM:eL" )

    a=pos_str(line1,"::",0);
    if (a>-1)
    {
      eC_exist=TRUE;
      leng=strlen(line1);
```

```
      c=pos_str(line1,"::",0);
      strcpy(line2,line1);
      in_delete(line2,c,leng-c);

      if (strcmp(line2,NULL)==0)
      {
      strcpy(line2,"NULL");
      }
      strcpy(eC,line2); //get eC

    } //if a

    if (eC_exist==TRUE)
    {
      strcpy(line2,line1);
      leng=strlen(line2);
      a=pos_str(line2,"::",0);
      in_delete(line2,0,a+2);  //line2 is "eM:eL" now
      strcpy(line1,line2);
      eC_exist=FALSE;
    } //if
    else
      strcpy(line2,line1);

    b=pos_char(line2,':',0);
    if (b>-1)
    {
      eM_exist=TRUE;
      leng=strlen(line2);
      c=pos_char(line2,':',0);
      in_delete(line2,c,leng-c);

      if (strcmp(line2,NULL)==0)
      strcpy(line2,"NULL");
      strcpy(eM,line2); //get eM
    } //if b

    if (eM_exist==TRUE)
    {
      b=pos_char(line1,':',0);
      leng=strlen(line1);
      in_delete(line1,0,b+1); //line1 is eL now
      eM_exist=FALSE;
    }

    if (strcmp(line1,NULL)==0)
    {
      strcpy(line1,"-1");
    }
    strcpy(eL,line1); //get eL now

    }    //if l!=k+1 ,is not just "probing()"
    else //just "probing()"
    {
      strcpy(eC,"NULL");
      strcpy(eM,"NULL");
      strcpy(eL,"-1");
    }
  } //if k

strcpy(line1,NULL);
```

```
strcpy(line2,NULL);

k=pos_char(line0,')',0);
if (k>-1)
{
  if (m=pos_char(line0,'{',0)>-1)
    l=pos_char(line0,'{',0);
  else
    l=pos_char(line0,';',0);

  if (l!=k+1)
  {
    strcpy(line1,line0);
    in_pos_mid(line1,k+1,l-1); //line1 is Boolean and pL
    strcpy(line2,line1);
    a=pos_char(line2,':',0);

    if (a>-1)
    {
    probingBoolean_exist=TRUE;
    leng=strlen(line2);
    c=pos_char(line2,':',0);
    in_delete(line2,c,leng-c);

    if (strcmp(line2,NULL)==0)
    {
      strcpy(line2,"No_Probing_Firing");
    }
    strcpy(probingBoolean,line2);
    } //if a

    if (probingBoolean_exist==TRUE)
    {
    b=pos_char(line1,':',0);
    leng=strlen(line1);
    in_delete(line1,0,b+1);
    probingBoolean_exist=FALSE;
    }

    if (strcmp(line1,NULL)==0)
    strcpy(line1,"-1");
    strcpy(pL,line1); //get pL

  }//if l

} //if k

/* comment statements include "probing"*/
i=pos_str(line,"probing",0);
in_insert(line," /*",i);
/* two possible ends of probing statement: ";" and "{" */
if ( pos_char(line,'{',0)>-1 )
{
  j=pos_char(line,'{',0);
  in_insert(line,"*/\n\t\t",j);
}
else
{
  j=pos_char(line,';',0);
  in_insert(line,"*/",j);
}
```

```
fprintf(outfile_var,"%s",line);

/*output to extension class file*/
if (strcmp(eC,"NULL")!=0)
{
  strcat(es,"\"");
  strcat(es,eC);
  strcat(es,"\", ");
}
else
{
  strcat(es,eC);
  strcat(es,", ");
}

if (strcmp(eM,"NULL")!=0)
{
  strcat(es,"\"");
  strcat(es,eM);
  strcat(es,"\", ");
}
else
{
  strcat(es,eM);
  strcat(es,", ");
}

strcat(es,"\"");
strcat(es,eL);
strcat(es,"\", ");

strcat(es,"\n\t\t");

strcat(es,"\"");
strcat(es,probingBoolean);
strcat(es,"\", ");

strcat(es,"\"");
strcat(es,pL);
strcat(es,"\", ");

if (strcmp(ClassName,"NULL")!=0)
{
  strcat(es,"\"");
  strcat(es,ClassName);
  strcat(es,"\", ");
}
else
{
  strcat(es,ClassName);
  strcat(es,", ");
}

if (strcmp(functionName,"NULL")!=0)
{
  strcat(es,"\"");
  strcat(es,functionName);
  strcat(es,"\" ");
}
else
```

```
  {
    strcat(es,functionName);
  }

  strcat(es,");\n");
  strcat(es,"\t  if(p.isEqual(x)){\n\t    ");
  strcat(es,ClassName);
  strcat(es," a");
  strcat(es,ClassName);
  strcat(es,";\n");

  if (strcmp(probingBoolean,"No_probing_Firing")!=0)
  {
    strcat(es,"\t    if (a");
    strcat(es,ClassName);
    strcat(es,".");
    strcat(es,probingBoolean);
    strcat(es,")");
  };

  strcat(es,"\t    {\n");
  strcat(es,"\t      a");
  strcat(es,ClassName);
  strcat(es,".");
  strcat(es,functionName);
  strcat(es,"();\n");
  strcat(es,"\t    }\n");
  strcat(es,"\t  }\n");
  strcat(es,"\t} \n");

  strcat(longstring,es);
  fprintf(extensionfile_var,"%s",longstring);

  fgets(line,80,infile_var);
} //treat_probing()


translator::treat_function(STRING ClassName, STRING FunctionName)
{
  STRING parameter0, parameter1;
  STRING extensionClassname;      //pC
  STRING extensionFunctionname;   //pM
  STRING parameter;               //pL

  STRING extensionBoolean;        //boolean

                          //ClassName: eC
                          //FunctionName: eM
  STRING extensionLabel;          //eL

  STRING longstring;
  STRING line0;

  strcpy(parameter,"-1");

  strcpy(parameter0,"NULL");
  strcpy(parameter1,"NULL");

  strcpy(extensionClassname,"NULL");
  strcpy(extensionFunctionname,"NULL");
  strcpy(longstring,"{ ExtensionProbing x(");
```

```
int j,i,k,l,m,n;
int leng = 0;
int a,b,c;
boolean extensionClassname_exist=FALSE;
boolean extensionFunctionname_exist=FALSE;
boolean extensionBoolean_exist=FALSE;
boolean endFunction=FALSE;

while( (endFunction==FALSE)&&
     (((m=pos_char(line,'}',0))== -1 )||(inlineFunction==TRUE)) )
/*while not end of function*/
{
  i=pos_str(line,"extension",0);
  if (i>-1) /*find "extension" in this line*/
  {
    strcpy(extensionClassname,"NULL");
    strcpy(extensionFunctionname,"NULL");
    strcpy(parameter,"-1");

    strcpy(extensionBoolean,"No_Extension_Firing");
    strcpy(extensionLabel, "-1");

    strcpy(longstring,"\n   { ExtensionProbing x(");

    n=pos_char(line,';',i);
    if (n!=i+9)                 /* is not just "extension;" */
    {
    k=pos_char(line,'(',i); /*begin of parameter*/
    if (k>-1)
    {
      l=pos_char(line,')',k); /*end of parameter*/

      if (l!=k+1)  /*is not just extension() */
      {
        strcpy(parameter,line);
        in_pos_mid(parameter,k+1,l-1); /*get parameter*/

        strcpy(parameter1, parameter);
        strcpy(parameter0, parameter);

        a=pos_str(parameter1,"::",0);
        if (a>-1)
        {
          extensionClassname_exist = TRUE;
          leng=strlen(parameter1);
          c=pos_str(parameter1, "::",0);
          in_delete(parameter1,c,leng-c);

          //don't use NULL, use "" in CS !!!!!!!!!!!!!

          if (strcmp(parameter1,NULL)==0)
          {
          strcpy(parameter1,"NULL");
          }

          strcpy(extensionClassname,parameter1); /*class name*/

        } /*if a*/

        if (extensionClassname_exist == TRUE)
```

```
    {
      strcpy(parameter0,parameter);

      leng=strlen(parameter0);
      a=pos_str(parameter0,"::",0);
      in_delete(parameter0,0,a+2); /* " ::" */

      /*parameter0 include func name and parameter number now*/

      strcpy(parameter1,parameter0);
      extensionClassname_exist = FALSE;
    } /*if*/

    b=pos_char(parameter1,':',0);

    if (b>-1)
    {
      extensionFunctionname_exist = TRUE;

      leng=strlen(parameter1);
      c=pos_char(parameter1,':',0);
      in_delete(parameter1,c,leng-c);

      if (strcmp(parameter1,NULL)==0)
      {
      strcpy(parameter1,"NULL");
      }

      strcpy(extensionFunctionname,parameter1);/*function name*/
    } /*if b*/

    if (extensionFunctionname_exist == TRUE)
    {
      strcpy(parameter1,parameter0);

      b=pos_char(parameter0,':',0);
      leng=strlen(parameter0);
      in_delete(parameter0,0,b+1);  /* ':' */

      extensionFunctionname_exist = FALSE;
    } /*if*/

    if (strcmp(parameter0,NULL)==0)
    {
      strcpy(parameter0,"-1");
    }

    strcpy(parameter,parameter0);/*parameter is number now*/

  } /*if l!=k+1,  not just "extension()" */
} /*if k, end of parameter */

strcpy(parameter0,NULL);
strcpy(parameter1,NULL);

k=pos_char(line,')',i);
if (k>-1)
{
  l=pos_char(line,';',k);

  if (l!=k+1)
```

```
        {
          strcpy(parameter0,line);
          in_pos_mid(parameter0,k+1,l-1);//get Boolean and
Extension_Label

          strcpy(parameter1,parameter0);
          a=pos_char(parameter1,':',0);
          if (a>-1)
          {
            extensionBoolean_exist=TRUE;
            leng=strlen(parameter1);
            c=pos_char(parameter1,':',0);
            in_delete(parameter1,c,leng-c);

            if (strcmp(parameter1,NULL)==0)
            {
            strcpy(parameter1,"No_Extension_Firing");
            } //other paremeter1 is Boolean expression string

            strcpy(extensionBoolean,parameter1);
          } // if a

          if (extensionBoolean_exist==TRUE)
          {
            strcpy(parameter1,parameter0);
            b=pos_char(parameter0,':',0);
            leng=strlen(parameter0);
            in_delete(parameter0,0,b+1);
            extensionBoolean_exist=FALSE;
          }

          if (strcmp(parameter0,NULL)==0)
          {
            strcpy(parameter0,"-1");
          }

          strcpy(extensionLabel,parameter0);

        } //if l!=k+1 :  Exension_Label and Boolean exist
      } /*if k*/

      } /* if isn't just "extension;" 10 letters*/

      in_insert(line," /* ",i);
      l=pos_char(line,';',k);
      in_insert(line," */ ",l);

      if (strcmp(ClassName,"NULL")!=0)
      {
      strcat(longstring,"\"");
      strcat(longstring,ClassName);
      strcat(longstring,"\", ");
      }
      else
      {
      strcat(longstring,ClassName);
      strcat(longstring,", ");
      }

      if (strcmp(FunctionName,"NULL")!=0)
      {
```

```
strcat(longstring,"\"");
strcat(longstring,FunctionName);
strcat(longstring,"\", ");
}
else
{
strcat(longstring,FunctionName);
strcat(longstring,", ");
}

strcat(longstring,"\"");
strcat(longstring,extensionLabel);
strcat(longstring,"\", ");

strcat(longstring, "\n\t\t");

strcat(longstring,"\"");
strcat(longstring,extensionBoolean);
strcat(longstring,"\", ");

strcat(longstring,"\"");
strcat(longstring,parameter);
strcat(longstring,"\", ");

if (strcmp(extensionClassname,"NULL")!=0)
{
strcat(longstring,"\"");
strcat(longstring,extensionClassname);
strcat(longstring,"\", ");
}
else
{
strcat(longstring,extensionClassname);
strcat(longstring,", ");
}

if (strcmp(extensionFunctionname,"NULL")!=0)
{
strcat(longstring,"\"");
strcat(longstring,extensionFunctionname);
strcat(longstring,"\"");
}
else
strcat(longstring,extensionFunctionname);

strcat(longstring,"); \n");

if(strcmp(extensionBoolean,"No_Extension_Firing")!=0)
{
strcat(longstring,"\t if (");
strcat(longstring,extensionBoolean);
strcat(longstring,") \n");
}

strcat(longstring,"         {\n");
strcat(longstring,"           Extension extension(x);\n");
strcat(longstring,"         };\n");

strcat(longstring,"    }");

in_insert(line,longstring,l+3);
```

```
    }/*if i, find "extension" in this line*/

    fprintf(outfile_var,"%s",line);

    if ( (inlineFunction==TRUE)&&(OneLine==TRUE) )
    {
      inlineFunction=FALSE;
      endFunction=TRUE;
    }
    else
    {
      if ((inlineFunction==TRUE)&&(m=pos_char(line0,'}',0)>-1))
      /* end of inline function */
      {
      inlineFunction=FALSE;
      endFunction=TRUE;
      }
      else
      endFunction=FALSE;
    }

    fgets(line,80,infile_var);
    strcpy(line0,line);

  }/*while*/

  if ((inlineFunction!=TRUE)&&(endFunction==FALSE))
    fprintf(outfile_var,"%s",line); /* end of function "};" */

} /*function treat_function*/
```

```
//filename: str.h

#ifndef STR_H
#define STR_H

#include <stddef.h>
#include <iostream.h>
#include "boolean.h"

class String
{
public:
  // constructor
  String();
  String(const String & str);
  String(const char * cstr);
  // destructor
  ~String();

  // value return methods
  size_t Length() const { return Len; };
  size_t Size() const  { return Siz; };

  // create a char-string from String method
  operator const char * () const { return Txt; };

  String operator = (const String & str);
  int operator == (const String & str) const {return (Compare(str) ==
1); };
  int Compare(const String & str) const;

  int pos_str(char *str, char *substr, unsigned int start_index);
  int pos_char(char *str, char ch, int start_index);
  void in_insert(char *str, char *substr, unsigned int index);
  int in_delete(char *str, unsigned int index, unsigned int count);
  int in_pos_mid(char *str, unsigned int first, unsigned int last);

  // character retrieval method
  char operator [] (size_t pos) const;

private:
  // instance variables
  size_t Siz; // allocated size
  size_t Len; // current length
  char * Txt; // pointer to text

  // class constant
  static size_t AllocIncr;

  // calc alloc size for needed bytes
  static size_t CalcSiz(size_t needed);
};

inline char String::operator [] (size_t pos) const
{
  if (pos >= Len)
    return '\x00';
  else
    return Txt[pos];
};
#endif
```

```
//filename: str.cpp

#include "str.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

//constant initialize
size_t String::AllocIncr = 8;

String::String()
{
  Len = 0;
  Siz = 0;
  Txt = NULL;
}

String::String(const String & str)
{
  Len = str.Len;
  Siz = str.Siz;
  if (str.Txt == NULL)
    Txt = NULL;
  else
  {
    Txt = new char[Siz];
    memcpy(Txt,str.Txt,Len + 1);
  }
}

String::String(const char * cstr)
{
  if ((cstr == NULL) || (cstr[0] == '\x00'))
  {
    Len = 0;
    Siz = 0;
    Txt = NULL;
  }
  else
  {
    Len = strlen(cstr);
    Siz = ((Len + AllocIncr) / AllocIncr) * AllocIncr;
    Txt = new char [Siz];
    memcpy(Txt,cstr,Len + 1);
  }
}

String::~String()
{
  if (Txt != NULL)
    delete Txt;
}

String String::operator = (const String & str)
//assignment
{
  Len = str.Len;
  Siz = str.Siz;
  if (Txt != NULL)
    delete Txt;
  if (Siz == 0)
```

```
    Txt = NULL;
  else
  {
    Txt = new char[Siz];
    memcpy(Txt,str.Txt,Len + 1);
  }
  return *this;
}

int String::Compare(const String & str) const
//compare
{
  if (Txt == NULL)
    if (str.Txt == NULL)
      return 1;  //equal: both NULL
    else
      return 0;  //not equal

  size_t count;
  if (str.Len < Len)
    count = str.Len;
  else
    count = Len;

  char   c1, c2;
  size_t i;
  for (i = 0; i < count; ++i)
  {
    c1 = Txt[i];
    c2 = str.Txt[i];

    if (c1 != c2)
    {
      if (c1 < c2)
      return 0;  //less
      else
      return 2;  //larger
    }
  }

  if (Len == str.Len)
    return 1;   //equal
  else
  {
    if (Len < str.Len)
      return 0;
    else
      return 2;
  }
}

int String::pos_str(char *str, char *substr, unsigned int start_index)
// search substr from start_index  position
{
     char *ptr;
     ptr=strstr((str+start_index), substr);
     if (ptr!=0)
          return (ptr-str);
     else
          return -1;
}
```

```
int String::pos_char(char *str, char ch, int start_index)
//search char ch from start_index  position
{
     int k;
     unsigned int n=strlen(str);
     for (k=start_index;((*(str+k)!=ch) && (k<n));k++);
     if (k==n) k=-1;
     return k;
}

void String::in_insert(char *str, char *substr, unsigned int index)
//insert substr into str from index position
{
     unsigned int m=strlen(substr);
     unsigned int n=strlen(str);
     if (index<n)
     {
          //int i;
          memmove((str+index+m),(str+index), n+1-index);
          memmove((str+index),substr,m);
     }
     else
          memmove((str+n),substr,m+1);
}

int String::in_delete(char *str, unsigned int index, unsigned int count)
//delete count number chars from index in str
{
     unsigned int n=strlen(str);
     if (index<0)
          index=0;
     if (index<n)
     {
        if ((index+count-1)>=n)
               str[index]='\0';
          else
     memmove((str+index),(str+index+count),n-index-count+1);
          return 0;
     }
     else
          return -1;
}

int String::in_pos_mid(char *str, unsigned int first, unsigned int last)
//get chars from first to last in str,
//    if return 0 then success
{
     unsigned int len=strlen(str);
     unsigned count;
     if (len==0||last<first)
          return -1;
     if (first>len) first=len;
     if (last>len) last=len;
     if (first>last) return -1;
     count=last-first+1;
     if (first>0)
     {
        in_delete(str,0,first);
        last-=first;
        len-=first;
```

```
    }
    if (last<len)
       in_delete(str,(last+1),(len-last));
    return 0;
}
```

```
//filename: try.c3

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

#include "EP.h"

class ClassA {
public:
      boolean eba2;
      boolean pba2;
      boolean pba3;
public:
      ClassA () { eba2=TRUE;
                  pba2=FALSE;
                  pba3=TRUE;
                };

      void am1( ) probing(){
            cout<<"am1 done! \n";
            };

        void am2( ) probing()pba2:;

      void am3( ) probing(main:)1;

};

void ClassA::am2( )
{
      cout<<"am2 done! \n";
      extension(ClassB::);
};

void ClassA::am3( )
{
      cout<<"am3 done! \n";
};

class ClassB {
public:
      boolean ebb1;
public:
      ClassB() {ebb1=TRUE;};
      void bm1( );
      void bm2( );
};

void ClassB::bm1( )
{
      cout<<"bm1 executed.\n";
};

void ClassB::bm2( )
{
      cout<<"bm2 executed.\n";
      extension(1)ebb1:0;
};
```

```
class ClassC {
public:
      boolean pbc1;
      boolean ebc2;
public:
      ClassC() {pbc1=TRUE; ebc2=TRUE;};

      void cm1( )probing(main:)pbc1:1{
                  cout<<"cm1 executed.\n";
            };

      void cm2( );
};

void ClassC::cm2( )
{
      cout<<"cm2 executed.\n";
      extension()ebc2:1;
};

void main()
{
      boolean ebma;
      boolean ebmb;

      ebma=FALSE;
        ebmb=TRUE;

      ClassA ca;
      ClassB cb;
      ClassC cc;

      extension(ClassC::)ebma:;
      extension(0)ebmb:;

      cb.bm2();
      cc.cm1();

        cout<<"done! \n\n";
};
```

```
//filename: try.cpp

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

#include "EP.h"

class ClassA {
public:
      boolean eba2;
      boolean pba2;
      boolean pba3;
public:
      ClassA () { eba2=TRUE;
                  pba2=FALSE;
                  pba3=TRUE;
                };

      void am1( )  /*probing()*/
            {
            cout<<"am1 done! \n";
            };

        void am2( )  /*probing()pba2:*/;
      void am3( )  /*probing(main:)1*/;
};

void ClassA::am2( )
{
      cout<<"am2 done! \n";
       /* extension(ClassB::) */
   { ExtensionProbing x("ClassA", "am2", "-1",
            "No_Extension_Firing", "-1", "ClassB", NULL);
         {
           Extension extension(x);
         };
   } ;
};

void ClassA::am3( )
{
      cout<<"am3 done! \n";
};

class ClassB {
public:
      boolean ebb1;
public:
      ClassB() {ebb1=TRUE;};
      void bm1( );
      void bm2( );
};

void ClassB::bm1( )
{
      cout<<"bm1 executed.\n";
};
```

```
void ClassB::bm2( )
{
      cout<<"bm2 executed.\n";
       /* extension(1)ebb1:0 */
   { ExtensionProbing x("ClassB", "bm2", "0",
            "ebb1", "1", NULL, NULL);
       if (ebb1)
         {
           Extension extension(x);
         };
   } ;
};

class ClassC {
public:
      boolean pbc1;
      boolean ebc2;
public:
      ClassC() {pbc1=TRUE; ebc2=TRUE;};

      void cm1( ) /*probing(main:)pbc1:1*/
            {
                  cout<<"cm1 executed.\n";
            };

      void cm2( );
};

void ClassC::cm2( )
{
      cout<<"cm2 executed.\n";
       /* extension()ebc2:1 */
   { ExtensionProbing x("ClassC", "cm2", "1",
            "ebc2", "-1", NULL, NULL);
       if (ebc2)
         {
           Extension extension(x);
         };
   } ;
};

void main()
{
      boolean ebma;
      boolean ebmb;

      ebma=FALSE;
        ebmb=TRUE;

      ClassA ca;
      ClassB cb;
      ClassC cc;

       /* extension(ClassC::)ebma: */
   { ExtensionProbing x(NULL, "main", "-1",
            "ebma", "-1", "ClassC", NULL);
       if (ebma)
         {
           Extension extension(x);
         };
   } ;
```

```
        /* extension(0)ebmb: */
    { ExtensionProbing x(NULL, "main", "-1",
              "ebmb", "0", NULL, NULL);
        if (ebmb)
          {
            Extension extension(x);
          };
    } ;

       cb.bm2();
       cc.cm1();

         cout<<"done! \n\n";
};
```

```
//filename: main.cpp
#include <iostream.h>
#include "boolean.h"
#include "ep.h"
#include "exten.h"
#include "ClassA.h"
#include "ClassB.h"
#include "ClassC.h"
void main()
{
      boolean ebma;
      boolean ebmb;

      ebma=FALSE;
        ebmb=TRUE;

      ClassA ca;
      ClassB cb;
      ClassC cc;

       /* extension(ClassC::)ebma: */
   { ExtensionProbing x(NULL, "main", "-1",
            "ebma", "-1", "ClassC", NULL);
       if (ebma)
         {
           Extension extension(x);
         };
   } ;
       /* extension(0)ebmb: */
   { ExtensionProbing x(NULL, "main", "-1",
            "ebmb", "0", NULL, NULL);
       if (ebmb)
         {
           Extension extension(x);
         };
   } ;

      cb.bm2();
      cc.cm1();

        cout<<"done! \n\n";
};
```

```
//filename: ClassA.h

#include <iostream.h>
#include "boolean.h"
#include "ep.h"

#ifndef ClassA_H
#define ClassA_H

class ClassA {
public:
      boolean eba2;
      boolean pba2;
      boolean pba3;
public:
      ClassA () { eba2=TRUE;
                  pba2=FALSE;
                  pba3=TRUE;
                };

      void am1( )  /*probing()*/
            {
            cout<<"am1 done! \n";
            };

        void am2( )  /*probing()pba2:*/;
      void am3( )  /*probing(main:)1*/;
};

#endif
```

```
//filename: ClassA.cpp

#include "ClassA.h"
#include "ep.h"
#include "exten.h"


void ClassA::am2( )
{
      cout<<"am2 done! \n";
       /* extension(ClassB::) */
   { ExtensionProbing x("ClassA", "am2", "-1",
            "No_Extension_Firing", "-1", "ClassB", NULL);
         {
           Extension extension(x);
         };
   } ;
};

void ClassA::am3( )
{
      cout<<"am3 done! \n";
};
```

```
//filename: ClassB.h

#include <iostream.h>
#include "boolean.h"
#include "ep.h"

#ifndef ClassB_H
#define ClassB_H

class ClassB {
public:
      boolean ebb1;
public:
      ClassB() {ebb1=TRUE;};
      void bm1( );
      void bm2( );
};

#endif
```

```
//filename: ClassB.cpp

#include "ClassB.h"
#include "ep.h"
#include "exten.h"


void ClassB::bm1( )
{
      cout<<"bm1 executed.\n";
};

void ClassB::bm2( )
{
      cout<<"bm2 executed.\n";
       /* extension(1)ebb1:0 */
   { ExtensionProbing x("ClassB", "bm2", "0",
            "ebb1", "1", NULL, NULL);
        if (ebb1)
          {
            Extension extension(x);
          };
   } ;
};
```

```
//filename: ClassC.h

#include <iostream.h>
#include "boolean.h"
#include "ep.h"

#ifndef ClassC_H
#define ClassC_H

class ClassC {
public:
      boolean pbc1;
      boolean ebc2;
public:
      ClassC() {pbc1=TRUE; ebc2=TRUE;};

      void cm1( ) /*probing(main:)pbc1:1*/
            {
                  cout<<"cm1 executed.\n";
            };

      void cm2( );
};

#endif
```

```
//filename: ClassC.cpp

#include "ClassC.h"
#include "ep.h"
#include "exten.h"


void ClassC::cm2( )
{
       cout<<"cm2 executed.\n";
        /* extension()ebc2:1 */
   { ExtensionProbing x("ClassC", "cm2", "1",
             "ebc2", "-1", NULL, NULL);
        if (ebc2)
          {
            Extension extension(x);
          };
   } ;
};
```

```
//filename: exten.h

#include "ep.h"
#include "ClassA.h"
#include "ClassB.h"
#include "ClassC.h"

#ifndef EXTEN_H
#define EXTEN_H

class Extension{
public:
      Extension(ExtensionProbing x);
}; //class Extension

#endif
```

```
//filename: exten.cpp

#include "exten.h"
#include "boolean.h"

Extension::Extension (ExtensionProbing x)
{

      {
      ExtensionProbing p(NULL, NULL, "-1",
            "No_probing_Firing", "-1", "ClassA", "am1" );
        if(p.isEqual(x)){
          ClassA aClassA;
          {
            aClassA.am1();
          }
        }
      }

      {
      ExtensionProbing p(NULL, NULL, "-1",
            "pba2", "-1", "ClassA", "am2" );
        if(p.isEqual(x)){
          ClassA aClassA;
          if (aClassA.pba2)       {
            aClassA.am2();
          }
        }
      }

      {
      ExtensionProbing p(NULL, "main", "-1",
            "No_probing_Firing", "1", "ClassA", "am3" );
        if(p.isEqual(x)){
          ClassA aClassA;
          {
            aClassA.am3();
          }
        }
      }

      {
      ExtensionProbing p(NULL, "main", "-1",
            "pbc1", "1", "ClassC", "cm1" );
        if(p.isEqual(x)){
          ClassC aClassC;
          if (aClassC.pbc1)       {
            aClassC.cm1();
          }
        }
      }
};
```

```
//filename: hash.h
#ifndef HASH_H
#define HASH_H

#include "persist.h"
enum HashKeyType
{
  KEY_STRING,
  KEY_LONG,
  KEY_INTEGER,
  KEY_OTHER
};

//HashEntryBase: Base class for entries in buckets ( 2 way linked lists
struct HashEntryBase
{
  HashEntryBase * Prev;
  HashEntryBase * Next;
  HashEntryBase();

  virtual HashKeyType GetKeyType() const = 0;
  virtual size_t Hash(size_t buckets) const = 0;
  virtual int KeyEquals(const HashEntryBase * entry) const = 0;
};

inline HashEntryBase::HashEntryBase()
{
  Prev = NULL;
  Next = NULL;
}

//HashEntryStr: A hashtable entry contains a KeyString
struct HashEntryStr : public HashEntryBase
{
  KeyString Key;

  HashEntryStr(const KeyString & k): Key(k) { };
  HashEntryStr(const HashEntryStr & e): Key(e.Key) { };

  virtual HashKeyType GetKeyType() const;
  virtual size_t Hash(size_t buckets) const;
  virtual int KeyEquals(const HashEntryBase * entry) const;

protected:
  HashEntryStr();
};

//HashEntryStrDB: A Hash entry contains a KeyString and a DataBlock
struct HashEntryStrDB : public HashEntryStr
{
  DataBlock Data;
  HashEntryStrDB(const KeyString & k,const DataBlock & db) :
HashEntryStr(k)
  {
    Data = db;
  };
};

//HashTableBase: Base class for all hash tables
class HashBucket;
class HashTableBase
```

```
{
public:
  HashTableBase(size_t buckets);
  ~HashTableBase();

protected:
  boolean AddEntry(HashEntryBase * newe);
  boolean DelEntry(const HashEntryBase * dele);
  boolean IsDupe(const HashEntryBase * dupe);
  const HashEntryBase * FindEntry(const HashEntryBase * finde);
  boolean Traverse();
  virtual boolean TravCallback(const HashEntryBase * e) const = 0;

  // data elements
  size_t NoOfBuckets;
  HashBucket * * Table;
};

//HashTravFunc: traversing hash table entries
typedef boolean(HashTableBase::*HashTravFunc)(const HashEntryBase*e)
const;

//HashBucket: Base class for buckets
class HashBucket
{
public:
  HashBucket();
  ~HashBucket();

  boolean AddEntry(HashEntryBase * newe);
  boolean DelEntry(const HashEntryBase * dele);
  boolean IsDupe(const HashEntryBase * dupe);
  const HashEntryBase * FindEntry(const HashEntryBase * finde);
  boolean Traverse(const HashTableBase & table, HashTravFunc func);

protected:
  HashEntryBase * First;
};

inline HashBucket::HashBucket()
{
  First = NULL;
}

//HashEnumFuncStrDB: retrieve all records in table
typedef boolean (*HashEnumFuncStrDB)(const KeyString & k,
                              const DataBlock & db);

//HashTableStrDB: A HashTable indexed by Strings
class HashTableStrDB : private HashTableBase
{
public:
  HashTableStrDB(size_t buckets): HashTableBase(buckets) { };

  boolean Insert(const KeyString & key, const DataBlock & db);
  boolean Delete(const KeyString & key);
  DataBlock LookUp(const KeyString & key);
  boolean Enumerate(HashEnumFuncStrDB func);

protected:
  virtual boolean TravCallback(const HashEntryBase * e) const;
```

```
  HashEnumFuncStrDB EnumCallback;
};

#endif
```

```
//filename: hash.cpp
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include "hash.h"

//HashEntryStr: A Hashtable entry contains a KeyString
HashKeyType HashEntryStr::GetKeyType() const
{
  return KEY_STRING;
}

size_t HashEntryStr::Hash(size_t buckets) const
{
  unsigned long n = 0;
  const String & str = Key.GetString();
  size_t len = str.Length();

  for (size_t i = 0; i < len; ++i)
  {
    n <<= 1;
    n += str[i];
  }

  return size_t(n % (unsigned long)buckets); //HASH function
}

int HashEntryStr::KeyEquals(const HashEntryBase * e) const
{
  if (KEY_STRING != e->GetKeyType())
    cout<<"mismatched types. \n";
  return (Key == ((const HashEntryStr *)e)->Key);
}

//HashTableBase: Base class for all hash tables
HashTableBase::HashTableBase(size_t buckets)
{
  // verify number of buckets
  if (buckets == 0)
    cout<<"cannot create 0 size table.\n";
  if (buckets < 9)
    cout<<"number of buckets is too small. \n";
  // store number of buckets
  NoOfBuckets = buckets;
  // allocate Table
  Table = new HashBucket * [NoOfBuckets];
  if (Table == NULL)
    cout<<"memory allocation failure.\n";

  // assign empty buckets to table
  for (size_t b = 0; b < NoOfBuckets; ++b)
  {
    Table[b] = new HashBucket;

    if (Table[b] == NULL)
      cout<<"memory allocation failure.\n";
  }
}

HashTableBase::~HashTableBase()
{
```

```
  // delete buckets
  for (size_t b = 0; b < NoOfBuckets; ++b)
    delete Table[b];
  // delete table
  delete[] Table;
}

boolean HashTableBase::AddEntry(HashEntryBase * newe)
{
  if (newe == NULL)
    return FALSE;
  size_t bucket = newe->Hash(NoOfBuckets);
  return Table[bucket]->AddEntry(newe);
}

boolean HashTableBase::DelEntry(const HashEntryBase * dele)
{
  if (dele == NULL)
    return FALSE;
  size_t bucket = dele->Hash(NoOfBuckets);
  return Table[bucket]->DelEntry(dele);
}

boolean HashTableBase::IsDupe(const HashEntryBase * dupe)
{
  if (dupe == NULL)
    return FALSE;
  size_t bucket = dupe->Hash(NoOfBuckets);
  return Table[bucket]->IsDupe(dupe);
}

const HashEntryBase * HashTableBase::FindEntry(const HashEntryBase *
finde)
{
  if (finde == NULL)
    return NULL;
  size_t bucket = finde->Hash(NoOfBuckets);
  return Table[bucket]->FindEntry(finde);
}

boolean HashTableBase::Traverse()
{
  boolean result;
  for (size_t n = 0; n < NoOfBuckets; ++n)
  {
    result = Table[n]->Traverse(*this,&HashTableBase::TravCallback);
    if (result == FALSE)
      break;
  }
  return result;
}

//HashBucket: Base class for buckets
HashBucket::~HashBucket()
{
  if (First != NULL)
  {
    HashEntryBase * e, * enext;
    e = First;
    while (e != NULL)
    {
```

```
      enext = e->Next;
      delete e;
      e = enext;
    }
  }
}

boolean HashBucket::AddEntry(HashEntryBase * newe)
{
  if (newe == NULL)
    return FALSE;
  if (First == NULL)
    First = newe;
  else
  {
    HashEntryBase * e = First;
    // search for last entry in list
    while (e != NULL)
    {
      // watch for duplicate keys
      if (e->KeyEquals(newe))
      {
      cout<<"duplicate key ignored.\n";
      return FALSE;
      }

      if (e->Next == NULL)
      {
      // link entry to end of list
      e->Next = newe;
      newe->Prev = e;
      break;
      }

      e = e->Next;
    }
  }
  return TRUE;
}

boolean HashBucket::DelEntry(const HashEntryBase * dele)
{
  if (dele == NULL)
    return FALSE;
  HashEntryBase * e = First;

  // search for key in list
  while (e != NULL)
  {
    // if key found, delete it
    if (e->KeyEquals(dele))
    {
      if ((e->Prev == NULL) && (e->Next == NULL))
      First = NULL;
      else
      {
      // remove entry from list
      if (e->Prev == NULL)
        First = e->Next;
      else
        e->Prev->Next = e->Next;
```

```
      if (e->Next != NULL)
        e->Next->Prev = e->Prev;
      }
      // delete entry
      delete e;
      // a success!
      return TRUE;
    }
    e = e->Next;
  }
  return FALSE;
}

boolean HashBucket::IsDupe(const HashEntryBase * dupe)
{
  if ((dupe == NULL) || (First == NULL))
    return FALSE;
  HashEntryBase * e = First;

  // search for key in list
  while (e != NULL)
  {
    // if key found, return it
    if (e->KeyEquals(dupe))
      return TRUE;
    e = e->Next;
  }
  return FALSE;
}

const HashEntryBase * HashBucket::FindEntry(const HashEntryBase * finde)
{
  if ((finde == NULL) || (First == NULL))
    return NULL;
  HashEntryBase * e = First;

  // search for key in list
  while (e != NULL)
  {
    // if key found, return it
    if (e->KeyEquals(finde))
      return e;
    e = e->Next;
  }
  return NULL;
}

boolean HashBucket::Traverse(const HashTableBase & table,
                             HashTravFunc func)
{
  boolean result;
  HashEntryBase * e = First;

  while (e != NULL)
  {
    result = (table.*func)(e);
    if (result == FALSE)
      break;
    e = e->Next;
  }
  return result;
```

```
}

//HashTableStrDB: Core class for hash tables
boolean HashTableStrDB::Insert(const KeyString & key,
                               const DataBlock & data)
{
  // create new HashEntryStrDB
  HashEntryStrDB * entry = new HashEntryStrDB(key,data);
  if (entry == NULL)
    cout<<"memory allocate failure.\n";
  // insert into table
  return AddEntry(entry);
}

boolean HashTableStrDB::Delete(const KeyString & key)
{
  // create new HashEntryStr
  HashEntryStr * entry = new HashEntryStr(key);
  if (entry == NULL)
    cout<<"memory allocation error.\n";
  // insert into table
  boolean result = DelEntry(entry);
  delete entry;
  return result;
}

DataBlock HashTableStrDB::LookUp(const KeyString & key)
{
  // create new HashEntryStr
  HashEntryStr * entry = new HashEntryStr(key);
  if (entry == NULL)
    cout<<"memory allocate error.\n";
  const HashEntryStrDB * e =
    (const HashEntryStrDB *)FindEntry(entry);
  delete entry;
  if (e == NULL)
    return NULL_BLOCK;
  else
    return e->Data;
}

boolean HashTableStrDB::Enumerate(HashEnumFuncStrDB func)
{
  if (func == NULL)
    return FALSE;
  EnumCallback = func;
  return Traverse();
}

boolean HashTableStrDB::TravCallback(const HashEntryBase * e) const
{
  if (e == NULL)
    return FALSE;
  else
  {
    HashEntryStrDB * e2 = (HashEntryStrDB *)e;
    return EnumCallback(e2->Key, e2->Data);
  }
}
```

```
//filename: persist.h
#ifndef PERSIST_H
#define PERSIST_H

#include "str.h"
#include "boolean.h"
#include <stddef.h>

class DataBlock
{
public:
  // constructors
  DataBlock();
  DataBlock(size_t sz, const void * data);
  DataBlock(const DataBlock & db);

  // destructor
  ~DataBlock();

  // assignment
  void operator = (const DataBlock & db);

  // interrogation
  size_t       GetSizeOf() const { return BufferSize; };
  const void * GetBufferPtr() const { return BufferPtr; };

  // check for NULL block
  boolean IsNull() const;

protected:
  size_t    BufferSize;
  void *    BufferPtr;
  static void ReportError();

};

inline DataBlock::DataBlock()
{
  BufferSize = 0;
  BufferPtr  = NULL;
}

inline DataBlock::~DataBlock()
{
  if (BufferPtr != NULL)
    delete BufferPtr;
}

inline boolean DataBlock::IsNull() const
{
  if (BufferSize == 0)
    return TRUE;
  else
    return FALSE;
}

extern const DataBlock NULL_BLOCK;

class KeyString
{
private:
```

```
    String KStr;

public:
  KeyString();
  KeyString(const String & str) : KStr(str) { };
  KeyString(const KeyString & keystr) : KStr(keystr.KStr) { };

  const String & GetString() const { return KStr; };
  virtual operator DataBlock() const;
  int Compare(const KeyString & key) { return KStr.Compare(key.KStr); };
  int operator == (const KeyString & key) const{return (KStr ==
key.KStr);};
};

class KeyByString
{
public:
  virtual KeyString MakeKey() const = 0;
};

#endif
```

```
//filename: persist.cpp
#include "persist.h"
#include <string.h>

const DataBlock NULL_BLOCK;

DataBlock::DataBlock(size_t sz, const void * data)
{
  BufferSize = sz;
  BufferPtr = (void *)new char[sz];
  if (data == NULL)
    memset(BufferPtr,0,sz);
  else
    memcpy(BufferPtr,data,sz);
}

DataBlock::DataBlock(const DataBlock & db)
{
  BufferSize = db.BufferSize;
  BufferPtr  = (void *)new char[BufferSize];
  memcpy(BufferPtr,db.BufferPtr,BufferSize);
}

void DataBlock::operator = (const DataBlock & db)
{
  BufferSize = db.BufferSize;
  if (BufferPtr != NULL)
    delete BufferPtr;
  BufferPtr = (void *)new char[BufferSize];
  memcpy(BufferPtr,db.BufferPtr,BufferSize);
}

KeyString::operator DataBlock() const
{
  return DataBlock((size_t)(KStr.Length() + 1),
                   ((const void *)((const char *)KStr)));
}
```

```
//filename: main.cpp
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include "hash.h"

//class Object
class Object: public KeyByString
{
public:
  Object(const String &key,const String &datas):Key(key),Datas(datas){};
  Object(const DataBlock & db);

  virtual operator DataBlock() const;
  virtual KeyString MakeKey() const { return KeyString(Key); };
  friend ostream & operator << (ostream & strm, const Object & obj);

private:
  String Key;
  String Datas;
};

Object::Object(const DataBlock & db)
{
  char*ptr=(char*)db.GetBufferPtr();
  Key = ptr;
  ptr+=Key.Length()+1;
  Datas=ptr;
}

Object::operator DataBlock() const
{
  size_t blen = Key.Length()+Datas.Length()+2;
  char*blk=new char[blen];

  if (blk==NULL)
  cout<<"\acannot allocate space for new object!";
  //store key
  strcpy(blk,Key);
  //store datas
  strcpy(blk+Key.Length()+1, Datas);

  return DataBlock(blen,(void *)blk);
}

ostream & operator << (ostream & strm, const Object & obj)
{
  strm<<obj.Key<<" of "<<obj.Datas;
  return strm;
}

boolean ShowThem(const KeyString &key,const DataBlock &db);

//main
int main()
{
  const size_t objCount=4;
  Object obj[objCount]=
  {
```

```
    Object("NULL NULL -1 No_probing_Firing -1 ClassA am1","ClassA.am1"),
    Object("NULL NULL -1 pba2 -1 ClassA am2","ClassA.am2"),
    Object("NULL main -1 No_probing_Firing 1 ClassA am3","ClassA.am3"),
    Object("NULL main -1 pbc1 1 ClassC cm1","ClassC.cm1"),
  };

  //create hash table
  HashTableStrDB ht(20);

  size_t n;
  //insert elements of obj into ht
  for (n=0;n<objCount;++n)
    ht.Insert(obj[n].MakeKey(), obj[n]);
  //display all entries in the table
  ht.Enumerate(ShowThem);

  //look for entry in the table
  String name1("NULL NULL -1 pba2 -1 ClassA am2");
  KeyString key1(name1);
  DataBlock db1(ht.LookUp(key1));

  //if db1 is null block, key1 wasn't found
  if (db1.IsNull())
  {
    cout<<"\aCould not find "<<name1<<endl;
    exit(EXIT_FAILURE);
  }

  //if key1 was found, create an OBject from the DataBlock
  Object obj1(db1);
  cout<<"\nFound: "<<obj1<<endl<<endl;

  //delete object with key "key1" from the tree
  if (FALSE == ht.Delete(key1))
  {
    cout<<"\aCould not delete "<<name1<<endl;
    exit(EXIT_FAILURE);
  }
  getch();
  cout<<endl<<"DELETED: "<<name1<<endl<<endl;

  //show all data in hash table now
  ht.Enumerate(ShowThem);

  cout<<"\nEND\n";
  getch();
  return 0;
}

//HashEnumFuncStrDB     callback function
boolean ShowThem(const KeyString & key, const DataBlock & db)
{
  Object obj(db);
  cout<<"key = "<<key.GetString()<<"\tdata = "<<obj<<endl;
  return TRUE;
}
```

```
am1 done!
bm2 executed.
am1 done!
cm1 executed.
done!
```

Vita Auctoris

Mr. **Frank An** was born on June 5, 1971, in **P.R. China**. He graduated from high school in 1989. He got his Honors Bachelor's Degree of Computer Science, **National University of Defense Technology, P.R. China** in July 1993. He is currently a candidate for the Master's degree in Computer Science at **University of Windsor**.