EXTENDING AN OBJECT-ORIENTED DESIGN METHOD: A C++ EXTENSION FOR IDEF4

A Thesis

by

LI-TSUNG HSIEH

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

[illegible]
Richard J. Mayer
(Chair of Committee)

[illegible]
Chuan-Jun Su
(Member)

[illegible]
Jeffrey S. Smith
(Member)

[illegible]
William M. Lively
(Member)

[illegible]
Way Kuo
(Head of Department)

May 1994

Major Subject: Industrial Engineering

ABSTRACT

Extending an Object-oriented Design Method:
a C++ Extension for IDEF4. (May 1994)

Li-Tsung Hsieh, B.S., Tunghai University
Chair of Advisory Committee: Dr. Richard J. Mayer

This research introduces an object-oriented implementation design method - IDEF4/C++. IDEF4/C++ is an extension of the IDEF4 object-oriented design method that incorporates C++ language considerations and practice to provide guidance and structure to ease the transition from an IDEF4 conceptual design to its implementation in C++.

To guide the development of IDEF4/C++, three IDEF5 ontological models are built: (1) an ontology of general object-oriented concepts; (2) an ontology of the IDEF4 method concepts; and (3) an ontology of the C++ programming language. Together these ontologies form the conceptual foundation of this research effort. They also provide a formal platform for understanding the mappings between the terminology and primitive concepts in these domains.

Extensions included in the IDEF4/C++ are: (1) an extended method syntax; (2) a transformation heuristic for transforming an IDEF4 conceptual design to an IDEF4/C++ implementation specification; (3) an IDEF3 model of the IDEF4/C++ design process with design evolution configuration management; and (4) best practice guidelines for the application of IDEF4/C++, especially focusing on design reuse.

The thesis concludes with a discussion of an integrated framework for object-oriented system development. Without increasing the complexity of the IDEF4 method, IDEF4/C++ takes advantage of C++ language features and best practice experience to bridge the gap between the conceptual design phase and the implementation phase in a software development project.

獻給我的母親，謝素雪女士

To my Mom, Mrs. Su-Xue Hsieh.

ACKNOWLEDGMENTS

Sincere thanks goes to Dr. Richard J. Mayer, my advisor. Without whose intensive guidance and limitless patience, this research would not be possible. I thank him for bringing me into the wonderful world of system engineering and, he if anyone, is the IDEF4/C++ method's godfather. I also wish to express my appreciation to Dr. Chuan-Jun Su for his valuable suggestions on my work in the Knowledge Based Systems Laboratory. I also wish to thank Dr. Jeffrey S. Smith and Dr. William M. Lively, the other members of my committee, for their suggested clarifications and constructive comments which make this thesis a more complete document.

This thesis also owes much to many friends. Much credit goes to the other members of the IDEF4/C++ development team, David Browne, Sue Wells, and Jim. Together we conquered this challenge and I am deeply grateful for their contributions. Special thanks goes to David for his intelligent ideas and incredible knowledge in C++. I also would like to express my appreciation to Dr. Paula S. deWitte at the Knowledge Based Systems, Inc. for her moral and financial support of this research.

I am also grateful to my friends in the KBS lab. Thanks goes to Sun, Wu, CFD, Su, Eddie, Joe, and Mark. Those "day and night" discussions with them have been invaluable in my study. Special thanks to Jyh and Jack for whose precious time and generous input to both my academic study and personal growth.

Finally, I would like to express gratitude to Jake Stockton. I am indebted to Jake for his patient navigation through the English language and constant encouragement during my study-life overseas.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

LIST OF TABLES

Page

LIST OF FIGURES

LIST OF FIGURES (CONTINUED)

LIST OF FIGURES (CONTINUED)

CHAPTER I

INTRODUCTION

1.1 Introduction

In software engineering, the traditional software development process is usually referred to in terms of the "waterfall" model (Boehm 76). Though further refining works on the model have related to different levels of detailing, the three most generally identified phases are analysis, design and implementation. Each of these phases possesses discrete activities, has its own objectives, and is governed by distinct philosophy. However, in recent years, the introduction of object-oriented technology has blurred the distinct boundaries between them (Meyer 87)(Korson & McGregor 90). The object-oriented technique combines the principles of encapsulation, polymorphism, and inheritance to promote software reuse and to reduce downstream errors and maintenance efforts.

The object-oriented technology blurs the boundaries between these phases for several reasons. First, the elements (objects and their relationships) focused in each phase become more tightly connected. The objects and their relationships identified in the analysis phase cast a basic understanding of the problem domain. The design and implementation phases consequently follow this understanding and are based on these objects and relationships to conduct their own activities. Second, the system development activities are conducted as "modes of thought" rather than as sequential phases or iterations. The development team usually goes back and forth between modes of thought, performing tasks to refine the design, analysis and implementation, on the fly. The blur is especially obvious along the boundary

This thesis follows the style and format of *International Journal of Production Research.*

between the design and implementation phases (Meyer 87). An effective method must support the process of filling in the details from the analysis through the design specification and all the way to the working program. The promised benefits of object-orientation can only be obtained by integrating these activities into a seamless framework. This integration can also provide a paradigm for one of the goals of Computer Aided Software Engineering (CASE) - code generation from solution specification.

1.2 Motivation

Unfortunately, object-orientedness means different things to different people (Nelson 90). There is still no general and widespread agreement on the object-oriented model; different object-oriented programming languages support different notions of objects, such as those graphic-based object-oriented design methods and tools (Booch 91, Rumbaugh 91). A method or a programming language's object model is important because it determines the built-in semantics the method or the language understands and is able to enforce. This variety between methods and languages' object models brings out a problem while making the transition into the implementation phase from the design phase in the software development process (where the boundary blurs most). The object model that a design is based on may be different from the one that the implementing language supports. More specifically, the "object-oriented" features provided by a design environment may not be supported by the target implementing language or vice versa. Different languages might give different interpretations (implementation) to the same feature. The designer may not be aware of this "Tower of Babel" in object-orientedness until the implementation process begins. When encountering this problem, occasionally the designers are forced to either change their original designs to suit the intended implementing language or switch to a proper target language. As a result, this dilemma increases the software development time and cost, and decreases application's performance as well.

(a)

```
(defclass UniversityEmployee ()
   (
    (name  : accessor name)
    (department  : accessor department)
    (ssn  : reader ssn)))
```

```
(defclass Student ()
   (
    (name  : accessor name)
    (department  : accessor department)
    (student-ID  : accessor st-id)))
```

(b)

```
class UniversityEmployee {
protected:
   char *name;
   char *department;
   int ssn;
public:
   char *get_name();
   void put_name(char *);
   char *get_department();
   void put_department(char *);
   int get_ssn();
};
```

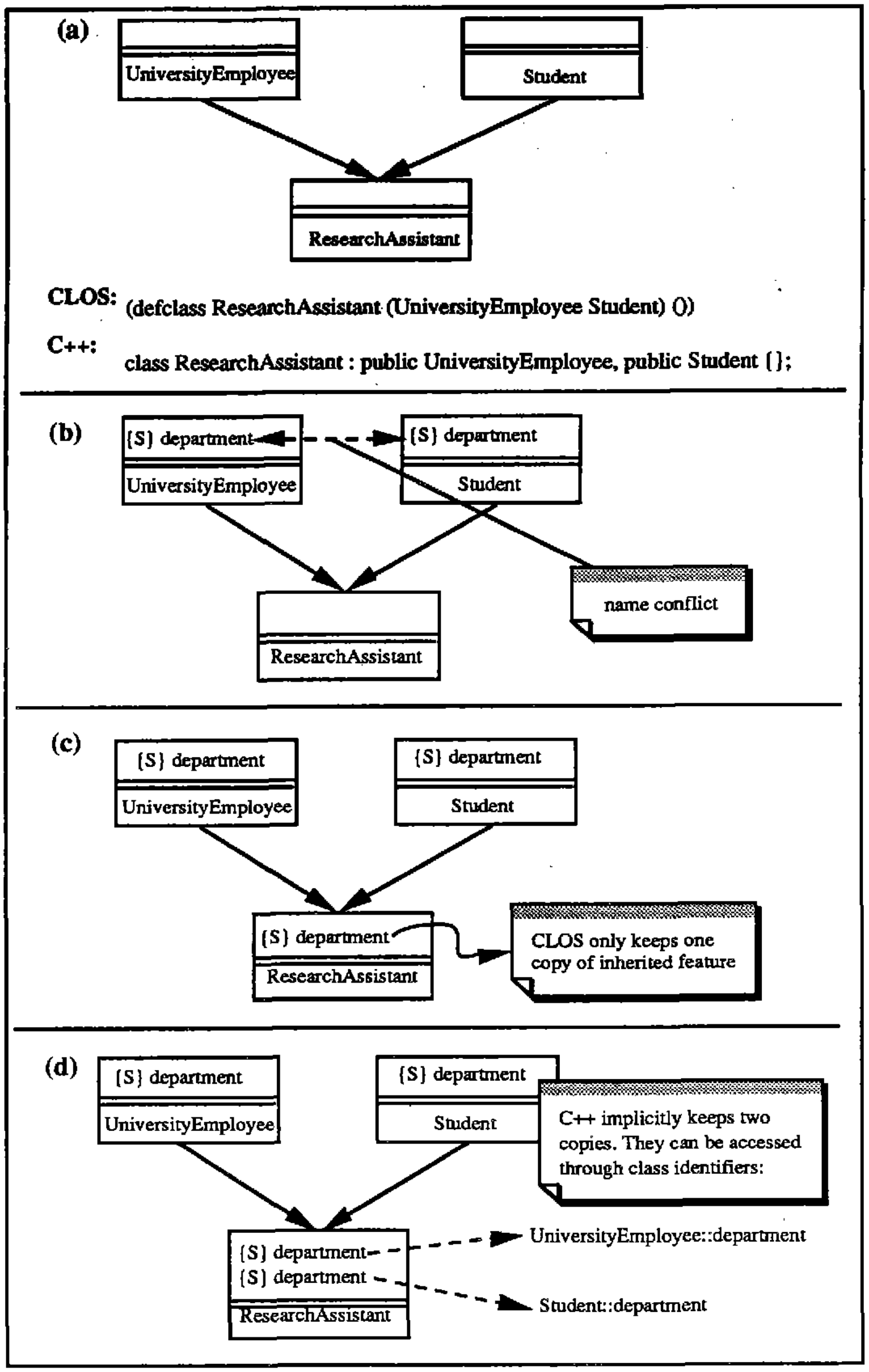
```
class Student {
protected:
   char *name;
   char *department;
   int student-ID;
public:
   char *get_name();
   void put_name(char *);
   char *get_department();
   void put_department(char *);
   int get_student_ID();
   void put_student_ID(int);
};
```

Figure 1-1. CLOS and C++ Code Examples of Name Conflicts.

For example, both the Common Lisp Object System (CLOS), an object-oriented extension of Common Lisp, and the C++ object-oriented programming language support multiple inheritance but implement it in somehow different way. Multiple inheritance, which allows a subclass to inherit features from more than one superclass, is straight-forward if no inherited features are multiply defined in the superclasses. But if more than one superclass has defined (or inherited) the same features, the language has to provide a strategy for resolving the name conflict occurred in the subclass. The strategy reflects the approach the language constructs and supports multiple inheritance, and conducts the method selection process as well. CLOS and C++ adopt different approaches of implementing this conflict resolving strategy. Basically, CLOS uses a class precedence list as a means for resolving the conflicts, which is in the order from most specific to least specific. On the other hand, C++ provides both "single-copy" and "multiple-copy" approaches[1]. For instance, consider that we have two classes: class *UniversityEmployee* and class *Student*, their declarations in CLOS and C++ are displayed in Figure 1-1(a) and (b) respectively. If we define a class ***ResearchAssistant*** as a subclass of both *UniversityEmployee* and *Student* as shown in Figure 1-2(a)[2], there will be a name conflict of the *department* slot because it is defined in both *UniversityEmployee* and *Student* (Figure 1-2(b)). CLOS uses a class precedence list to resolve the conflict and it will keep only one copy of *department* in *ResearchAssistant* (Figure 1-2(c)), whereas C++ keeps both copies of *department* implicitly (if the multiple-copy approach is applied). Both copies can be explicitly accessed by using class identifiers (Figure 1-2(d)). Because CLOS only keep one copy of *department*, the designer needs to determine which superclass is the *department* inherited from; does it refer to the research assistant's academic department or the department that hires him or her? Furthermore, the access to each of the *department* methods might be different and there might be conflicts in their contracts. Thus, it is necessary for the designer to decide which methods to be shadowed to hold the consistency.

[1] If the inheritance link is declared as `virtual`, then single-copy approach is adopted. Otherwise, multiple-copy is the default approach. See Section 6.8 for more details.

[2] The Figures are presented in terms of IDEF4/C++ notation. Where boxes represent classes and "S" symbol represents a slot feature. See Chapter 4 for more details about the IDEF4/C++ syntax.



```
(defclass ResearchAssistant (UniversityEmployee Student) ())
```

```
class ResearchAssistant : public UniversityEmployee, public Student {};
```

Figure 1-2. Different Name Conflict Resolutions of CLOS and C++.

It is not our intent to judge which language is better and which is not, as different languages are designed for different intents. For example, one of the primary goals of CLOS is to gain flexibility and extensibility for the language, whereas C++ is more focusing on run-time efficiency and implementational simplicity. For the example illustrated above, more design effort should be taken if CLOS is considered as the target implementing language[3].

The variety in interpretation (implementation) within the object-oriented paradigm therefore shadows the promise that the technique has proclaimed. Intuitively, the best solution is establishing a standard object model which every vendor of the implementing languages and the developer community would agree upon. Since object technology is still evolving, it is understandable that such a paradise will not appear soon. The need of extensibility for the design environment to support different implementing languages is therefore eminent.

One approach to solving the problem is to provide language-dependent extensions for a language-independent design method. The idea is straight-forward. Before the designer goes into the detailing mode (which is very related to the implementation language selected), the generic design environment should be able to be extended seamless in order to conduct this process efficiently.

More specifically, the design environment should be able to support the evolutionary change from the conceptual design all the way down to the implementation, smoothly and elegantly. Our intent is to construct an extensible environment for designers to be able to move through the process seamlessly. In addition, if an intended implementation language is found not to be expressive enough for the specific design, by using a certain mechanism provided in the design

[3] Here we only give a simplified example. However, for a more complicate design, there might be some compromise to the reality and tradeoffs between picking up different approaches.

environment, the designer can backtrack to the generic design and evolve and extend the design towards another target language.

1.3 Research Goal and Objectives

The specific goal of this research is to construct a C++ extension for IDEF4, an object-oriented design method, as a specialized design environment for the C++ implementation of a system. To achieve this goal, several objectives are identified. We group these objectives into the following:

1. Analyze and understand the domain:
   - To capture the ontology of object-orientation.
   - To capture the ontology of the implementation independent IDEF4 object-oriented design method.
   - To capture the ontology of the C++ object-oriented programming language.
2. Design and develop the extended IDEF4/C++ method:
   - To develop a C++ extension of IDEF4 method.

1.4 Organization of the Thesis

The results of this research are presented and organized as follows:

Chapter I introduces the object-oriented software development process, and identifies the evidence of various dialects among the object-oriented society, which motivates the activity of this research work. The research objectives is stated in the chapter as well.

Chapter II reviews related literature and presents the ontologies of the domain, which include core concepts and terminology used to conduct the research.

Chapter III discusses the basic concepts of the IDEF4 object-oriented design method family (IDEF4 and IDEF4/C++). The discussion is intended to lay out a foundation for the succeeding chapters.

Chapter IV presents the syntax of the extended method - IDEF4/C++. A number of examples and C++ code are given along with the introduction of the notations.

Chapter V specifies the IDEF4/C++ design development procedure. An IDEF3 dynamic model describing the design process with multiple developers / development teams is also presented.

Chapter VI discusses the principles and techniques of the use of the method. The discussion focuses on issues of design with reuse in IDEF4/C++. We conclude the chapter by summarizing rules of thumb for reuse.

Chapter VII gives the conclusion drawn from this thesis. Future extensions of the research are also discussed.

CHAPTER II

BACKGROUND

2.1 Literature Review

Both (Meyer 87) and (Korson & McGregor 90) mentioned the blurring between design and implementation phases in the object-oriented software development. Korson and McGregor suggest that the transition from design to implementation should be smooth and this transformation should be part of the design process. Meyer suggests that this is basically a technical problem (Meyer 88) and goes on to present a tool named Eiffel, claiming that Eiffel is a language for both design and implementation.

However, there is a major distinction between the philosophy and design intention of IDEF4 and those of Meyer's that we would like to address in the first place. We consider design and implementation[4] as different activities, whereas Meyer places the design process only at a higher level of abstraction than implementation (Meyer 88). Design and implementation are different in terms of the notion of correctness (Mayer 90). More specifically, the aim of the implementation activity (the programming activity) is to produce a specific single executable implementation which will run correctly and ultimately bug-free. This is referred as *P-Correct*. On the other hand, the aim of the design activity is to narrow the range of available choices so as to expedite the eventual acquisition of a correct solution for the problem. This is referred as *S-Correct*. Figure 2-1 (Mayer 90) describes the difference in the refinement process in these two activities. This figure shows that the objective of design evolution is to gain a larger probability for

[4] The term implementation used here mainly refers to the activity of programming.

correct implementation[5]. Based on this philosophy, unlike Meyer's approach (Eiffel), we choose to extend an existing logical design method into a language dependent implementation design method. That is, we propose extending the generic IDEF4 to IDEF4/C++, rather than turning a programming language into a design tool.

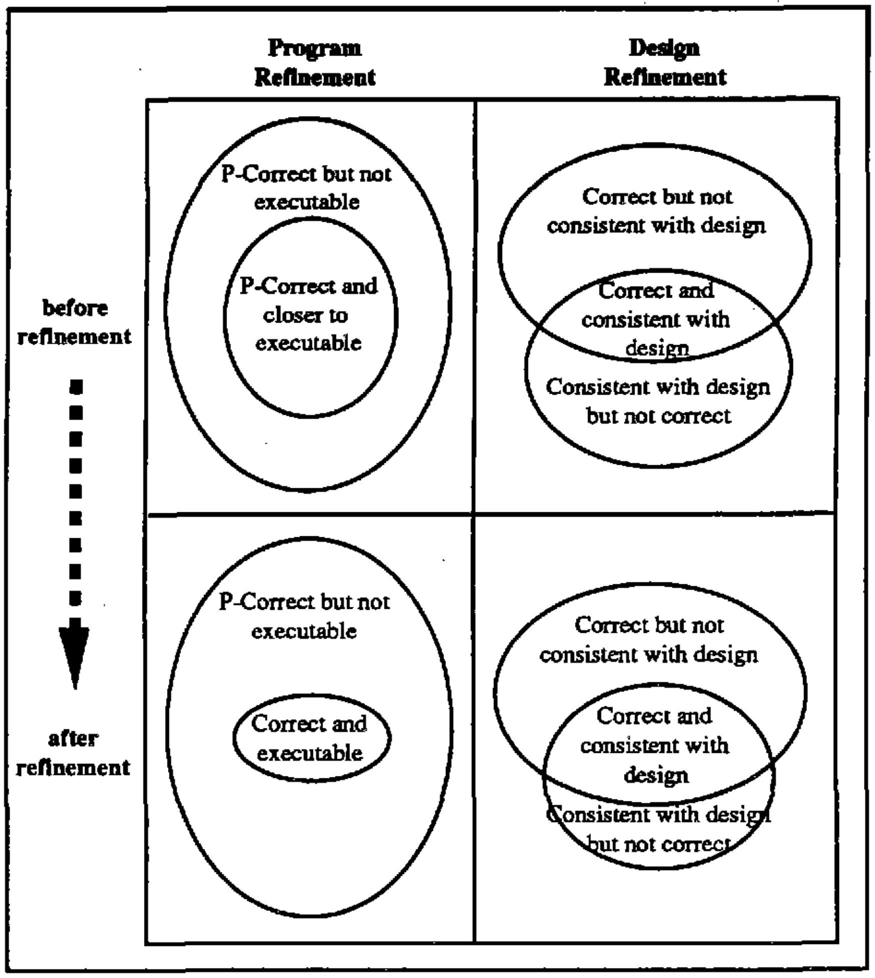

Program Refinement
Design Refinement
before refinement
after refinement
P-Correct but not executable
P-Correct and closer to executable
Correct but not consistent with design
Correct and consistent with design
Consistent with design but not correct
P-Correct but not executable
Correct and executable
Correct but not consistent with design
Correct and consistent with design
Consistent with design but not correct

Figure 2-1.
Design Refinement and Implementation Refinement: P-Correct and S-Correct.
Reprinted from (Mayer 90).

[5] The larger the S-Correct proportion of the design means the more chances to derive a correct implementation from the design model.

(Nelson 90) discusses the subject of variations in object-orientedness. He urges an agreement on this area, at least in the basic terminology. Alan Snyder has developed a common framework for general object-oriented terminology. He proposes an abstract object model (Snyder 90 and 93) which has a good organization on the basic concepts and terminology. Relating to this work is the specification (OMG 91) prepared by the Object Management Group, which makes a promise to become the standardization in this field.

Several object-oriented design methods including (Booch 91, Coad & Yourdon 91, and Rumbaugh 91) all have comprehensive discussions of the field. We include the survey on these methods to gain a broader understanding of object-oriented design models. (Stroustrup 90), (Lippman 91), and (Coplien 92) give a thorough overview on the C++ language, where issues such as type system, multiple inheritance, dynamic binding, function overloading / overriding, class template etc., are addressed and discussed in these sources. (DeMichiel 93) compares the distinction between Common Lisp Object System (CLOS) and C++, which provides some basic notions for understanding the design philosophy behind C++.

2.2 Domain Ontologies

Every method or language has an object model, which plays the role as the foundation for its notions of objects. The object model behind a method or a language is important because it provides the underlying constructs for specifying the built-in semantics that the method or the language understands and is able to enforce. Therefore, before extending the generic IDEF4 object-oriented design method to a specification for the C++ implementation purpose, the understanding (analysis) of the object models in the domain (IDEF4 and C++) is essential and necessary. This section presents the results of using IDEF5 ontological schematics for capturing the ontology models of the IDEF4 method and the C++ language, as well as the general concepts of object-orientedness. The concepts and syntax of IDEF5 method will not be discussed in the report, details of the method can be referenced in (Mayer 92c). However, major efforts lay in the process of "name-coining", which results in a set of core concepts for building up the target semantic models, and establishing the terminology to be used in descriptions of the domain.

Section 2.2.1 presents the model of general object concepts. Section 2.2.2 presents the IDEF4 object model and Section 2.2.3 presents the C++ object model. In these sections, only the summaries of the core concepts and key terms are presented.

2.2.1 Ontology of General Object-oriented Concepts

Establishing the common concepts and perceptions of an object-oriented model can greatly enhance the communication among object-oriented system developers, users, and researchers. In this section, we identify a general object model for such purpose, which is intended to provide an organized presentation of terminology and primitive concepts for the research work. IDEF4 (Mayer 92a), Booch's method (Booch 91), Rumbaugh's Object Modeling Technique (OMT) (Rumbaugh 91), and C++ (Stroustrup 90, Lippman 91 and Coplien 92) are all the sources for this ontology research. In addition, the ontology is also elaborated from the abstract object model used in the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) (OMG 91). The following summarizes the core concepts and the key terms identified from the above sources.

- An *object* is an identifiable, encapsulated entity that is capable of requesting and/or providing one or more services. It has *state*, is capable of performing some well-defined *behavior* and has an unique *identity* .

- The identity of an object is denoted by the term *handle*, which is a value that unambiguously identifies an object. The name of an object can be a handle to that object.

- The state of an object is captured in terms of a set of *attribute-value pairs* .

- The behavior of an object is captured in terms of a set of *operations* that the object can perform.

- A *feature* is a generic term for presenting a particular characteristic of the state or the behavior of an object.

- A feature can be an attribute or an operation.

- Packing related attributes and methods together is called *encapsulation.* Encapsulation protects data from corruption by other objects and hides low-level implementation details from the rest of the system.

- The mechanism for encapsulation is the object.

- Objects interact with each other in terms of *issuing requests*.

- An object that requests services is called a *client object*. An object that provides services is called a *server object*.

- A client object requests a service from a server object by issuing the request to the server object.
- A *request* is an event.
- A *service* is a computation that may be performed to respond to a request.
- The information associated with a request consists of an operation, a target server object, and zero or more parameters required to provide the service.
- An *operation* is specified to denote a service that can be requested. It has an associated *signature* which describes the *types* of the request parameters and return values.
- A *method* is an implementation of an operation. It is the code that may be executed / invoked to perform a requested service.
- A service is provided by means of performing an operation.
- An operation can be generic. A *generic operation* can be performed differently by different target objects by invoking different methods.
- A *binding* is a computation that results in the selection of the methods to perform a requested service. Binding can be *dynamic binding* or *static binding* according to the time that the decision is made.

- A *type* is a specified predicate defined over expressions to serve the purpose of membership checking or binding validation. Therefore, the type of an operation can be considered as the signature of the operation.

- A *class* is a specified template for defining attributes and operations for a particular type of objects. Objects of the same class have the similar sets of attributes and operations. Class can be instantiated to create objects.

- An *interface* of a class is a description of operations and attributes defined in that class. It includes the signatures of the operations, and the types of the attributes.

- Every class is a type, but not every type is a class. Type classifies objects according to a common interface they share. Therefore, the type of an object can be considered as the interface of the class of that object.

- Class *inheritance* is a specification of class definitions based upon the generalization / specialization relations between classes. Inheritance can be multiple inheritance or single inheritance.

- Object *aggregation* is a relationship in which one object is composed of other objects.

2.2.2 Ontology of the IDEF4 Method Concepts

In this section, we summarize the ontology of the object model of the generic IDEF4. The ontology is presented in terms of a set of the key concepts and coined terminology described as follows.

- An IDEF4 model consists of two submodels: a *Class submodel* and a *Method submodel*. These submodels are connected by means of a mechanism called *Dispatch Mapping*.

- The IDEF4 class submodel is composed of *Class Lattice Diagram*s, *Inheritance Diagram*s, *Type Diagram*s, and *Instantiation Diagram*s.

- The IDEF4 method submodel is composed of *Method Taxonomy Diagram*s and *Client Diagram*s.

- Each class in IDEF4 is associated with a *Class Invariant Data Sheet* (*CIDS*), which specifies the constraints for the instances of this class. Information such as direct present *feature*s, superclasses and subclasses are also documented here. The corresponding inheritance diagrams and type diagram of a class are referenced on its CIDS.

- In IDEF4 the term *feature* is used as a generic term to refer to both *attribute*s and *routine*s. Attributes denote value-returning features. Routines denote computation-initiating features.

- A routine can be refined as a *procedure* or a *function* in the late (detailing) stages of a design. An attribute can be refined as a function or a *slot*.
- Routines (functions and procedures) are behavioral features; they denote the behaviors of the instances of a class. Behavioral features are listed in a class invariant data sheet (CIDS) according to the generic behaviors they identify.
- Inheritance relationship between classes are described in inheritance diagrams.
- Access of the features presented in a class can be *public* or *private*; public features are accessible to other classes, private features are accessible only to the owner class and all its subclasses.
- A method taxonomy diagram describes a generic behavior.
- A method taxonomy diagram classifies a generic behavior into several *method sets* according to the similarity of the constraints on them. Method sets in a method taxonomy diagram are arranged in a more specific order from left to right or from top to bottom.
- Each method set in method taxonomy diagrams is associated with a *Contract Data Sheet* (*CDS*), which specifies the constraints that the implemented methods in this method set should satisfy.
- A *contract* is a set of constraints for the associated method set.

- *Methods* in a method set must be implemented according to the contract associated with the method set.
- Behavioral features defined in the class submodel and method sets in the method submodel are mapped with each other through dispatch mapping.
- Conflicting constraints are those constraints that redefine or shadow the constraints from previous method sets. Non-conflicting constraints are sets of pre- or post-conditions that should be applied with the inherited constraints.
- Type diagrams describe the aggregation relationship between classes. Type links are used to represent the relationship.
- The type of a feature in a class is specified as the class that is connected by a type link. Predefined types are collected in the *User Predefined Data Type List* associated with the method submodel.
- Only value-returning features, such as functions and slots can be shown in type diagrams.
- Type links have different kinds. Type links can be no inverse, with inverse, or with partial inverse.
- A client diagram describes the internal algorithmic structure (i.e., subroutine calls) of a behavioral feature. In a client diagram, a supplier routine is called by a client routine.

- Routines in client diagrams usually are shown with their defining classes. If a class associated with the routine is not specified, then a dynamic (run time) binding will occur in the implementation.
- Class lattice diagrams provide a broad view for the lattice of either the whole class submodel or the focused part of the submodel.
- Instantiation diagrams are associated with type diagrams in the class submodel. Instantiation diagrams describe the anticipated situations of composite links between instantiated objects that are used to validate the design.

2.2.3 Ontology of the C++ Programming Language

This section presents a set of key terms and the core concepts for the ontology of C++. We want to emphasize that the ontology is discovered and documented with the intent of only mapping significant characteristics of the language to the primitive object-oriented concepts; rather than focusing on the language syntax or structure.

- C++ *derived class* is a synonym of "subclass" in general object terminology.
- C++ *base class* is a synonym of "superclass" in general object terminology.

- C++ provides three feature accesses: *public*, *private*, and *protected*. Public features can be accessed by the whole system. Private features can be accessed only by the owner class. Protected features can be accessed by the owner class and its derived classes.

- A derived class inherits those non-private features defined in the base class.

- Private features in base classes can not be inherited by the derived classes. Private feature access control provides a means for implementing encapsulation.

- *Derivation* in C++ means inheritance. C++ provides three types of derivations: *public*, *private*, and *protected.*

- Non-private features of the base class become protected features of the derived class in a protected derivation.

- Non-private features of the base class become private features of the derived class in a private derivation.

- In a public derivation, a derived class inherits a base class's non-private features without changing their feature access.

- A *virtual* derivation in a multiple inheritance is used to prevent the name conflicts of the inherited features.

- Features of a class are called class *member*s in C++. Members in C++ classes are *data member*s or *member functions*.

- C++ data members implement the slots defined in a class.
- C++ member functions implement the functions or procedures in a class.
- C++ *virtual function overriding* is an example of dynamic polymorphism, which is also called run-time polymorphism or late binding. The invocation of a function is determined at run-time.
- C++ *function overloading* is an example of static polymorphism, which is also called compile-time polymorphism or early binding. The invocation of a function is determined at compile-time.
- C++ class *template* declaration implements the concept of parameterized class.
- C++ *friend* declaration provides a means to break encapsulation (information hiding). Classes or functions declared as friends to a class can access not only the non-private features but also the private features of that class.
- C++ *static* class member declaration implements the use of class variables and class operations. Class variables and operations are the members that only keep one copy among all the instances of a class.
- *Pure virtual* member functions construct *abstract base classes*. Member functions declared as pure can have no function bodies

implemented, which prevents any creation of instances from an abstract base class.

- Nested classes are the classes defined in other classes' definitions. The visibility of a nested class is limited to the scope of its enclosing class.

Note that we use *request* instead of the traditional term *message* for several reasons. One major reason is that *message sending* implies concurrent execution by the client and server objects (between the sender and the receiver). However, The intent is not to present an unified object model nor to compare and judge the various dialects in the object society (by saying who's right and who's wrong). Instead, we carry out these ontological models to form the boundary for our research domain. In other words, these models together intend to provide and define the primitive object concepts and terminology that can be used in this research work; especially, a set of terms that we can use for communication. A glossary of the terminology identified is given in Appendix A.

CHAPTER III

METHOD CONCEPTS

3.1 Introduction

IDEF4 family are methods for object-oriented design; they are not object-oriented programming languages (OOPL). However, basic object concepts supported by either a design method or a programming language are similar. The major elements for constructing an object-oriented system are commonly identified as classes, features, and methods. These basic elements are incorporated into the IDEF4 method family and form the foundation of IDEF4 and IDEF4/C++. In this chapter, we will discuss these basic method elements and their representation in the methods as well. Basic concepts such as class and type, class and object, class inheritance, feature taxonomy, feature type, method taxonomy, contract, and method set are included in the discussion.

A good way of thinking of an object-oriented system is of a space which consists of a set of independent but cooperating objects. Each object has state and behaviors. The state of an object is captured by a set of attributes with values assigned, whereas the behaviors are actually implemented by a set of **methods**. In the system development process, these objects are to be classified into a set of "packages" according to the common state and behaviors that they possess. Both the state and behaviors are characterized by a set of **features** in the design evolution, and these "packages", in the common object-oriented terminology, are called **classes**. Object-oriented design and programming activities tend to define these features and methods for classes. However, a novice in object-orientedness will often confuse the term type, class, and object.

3.2 Classes

3.2.1 Classes vs. Types

Each class is a type, but not every type is a class. A class is specified by the definition of a set of local, state-defining attributes and of a set of methods that define the behaviors of the instances of that class and their relationship to the instances of other classes that make up the system. In other words, a class is a data structure that includes a set of state-defining attributes and a set of methods that apply to the instances of that class. A type, on the other hand, is specified by a predicate defined over a set of expressions to serve the purpose of type checking or operation binding. Many object-oriented languages have used run-time type checking to ensure that the requests that are sent to an object are understood by that object. The type of an operation is referred to as the signature of that operation (signature type). Generally, a class can be instantiated to create objects in the system, whereas type classifies objects in terms of the common interface of their defining classes[6]. In other words, the type of an object can be considered as the interface of the class which that object belongs to. In this context, one of the important properties of objects is the property of substitution (Atkins and Brown 91), which states that objects providing similar operations can be used exchangeably if only the common behavior is required. In this sense, inheritance - a mechanism in which subclasses possess common behaviors defined in superclasses, therefore suggests that the type of an object should be associated with its class, and that the instances of a subclass should be able to used in all the places where instances of the superclass are expected. In other words, this formalizes the concept that instances of a subclass are also instances of its superclass. Such a concept has formed the

[6] Recall that an interface of a class is a description of the operations and attributes defined in that class. It includes the signatures of operations and the types of attributes (Section 2.2.1).

basis of those strong-typed object-oriented languages, such as C++. C++ implements this concept in terms of type conversion, which enforces an efficient request dispatch scheme and implements dynamic polymorphism (see Section 6.5).

3.2.2 Classes vs. Objects

The self-referential definition of classes and objects, in which an *object* is defined to be an instance of a *class* and a *class* is defined to be a description of similar *objects*, is often confusing. The terms *class* and *object* are usually heavily overloaded in the object-oriented literature. In (Mayer 92a), the meanings that the term *class* may refer to are summarized into the following:

- categories, or types, of objects in the real world (real-world perspective);
- data types representing categories of objects (data-item perspective); and
- modules of associated operations that define data types (module perspective).

The meanings that the term *object* may refer to are summarized as:

- real-world objects (real-world perspective); and
- data items belonging to one class or another (data-item perspective).

Consequently, *class* and *object* are defined by each other. We clarify this confusion from the perspective of the system analysis and design processes. In the system analysis process, one of the objectives is to identify the real-world objects from the problem domain (real-world perspective for *object*). These real-world objects are then classified into classes (real-world perspective for *class*) in the design process. Extra characteristics (features or classes) may be added for constructing an object model which provides a solution to the problem (data-item and module perspectives for *class*). Applying the solution to the problem is therefore the process of instantiating and activating the instances from those model classes. The object instances existing in a computer that forms the solution domain are therefore referred to as model objects (data-item perspective for *object*).

In class-based systems, the analysis focuses on real-world *objects*, the design focuses on *classes* of model objects. The classes are fabricated, rearranged, or synthesized in the design process to form the solution model. Each class contains a set of feature definitions that characterize the state and behavior of the instances of that class. The set of feature definitions consists of attributes and methods. The attribute definitions are used by the instances of the class to store their state. The methods characterize the behavior of instances of the class.

3.2.3 Class Box

Classes are the major syntactic construct in the IDEF4 method family, as in all class-based object-oriented formalisms. In IDEF4 and IDEF4/C++, a class is represented by a square-cornered box (see Figure 3-1) with the name of the class listed below the double line at the bottom of the box. IDEF4 requires that the first letter of the class name be capitalized. The features of the class are also displayed in the Class Box with private features displayed below the export line and with public features displayed above the export line[7]. Various feature symbols, prefixed to the feature name displayed in the class box, may also be used to provide additional information about the role that the feature plays. For each class defined, IDEF4 method family allows the attachment of class-invariant constraints using class-invariant data sheet (CIDS). These class-invariant constraints represent additional information about the definition of a class that is true for all instances created by the class at all times. The class-invariants described in a design provide constraints on the implementation of the design and serve as part of the specifications for a class.[8]

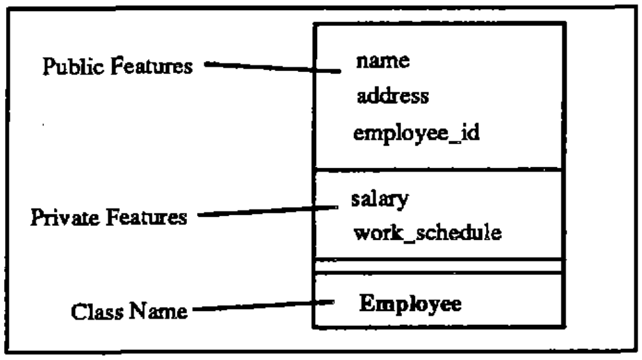


Figure 3-1. Class Box in IDEF4.

3.2.4 Class Inheritance

One of the most distinguishing characteristics of object technology is inheritance, especially multiple inheritance. Multiple inheritance allows a subclass to inherit features from more than one superclasses. The concept of inheritance

[7] IDEF4/C++ extends this representation with the addition of the display of protected features. See Chapter IV for more details.
[8] In an IDEF4/C++ implementation design, these CIDSs are the major sources for coding C++ class definitions. Section 4.3.6 gives the detailed discussion.

provides a means of organizing related classes into an inheritance hierarchy and supports for the reuse of methods and features in terms of subtyping (refer to Chapter VI for reuse by inheritance). The inheritance mechanism operates and follows the specialization/generalization relation. That is, the inheriting class (subclass) is a specialization of the class from which it inherits (superclass), and the inherited class (superclass) is a generalization of the class (subclass) that inherits it.

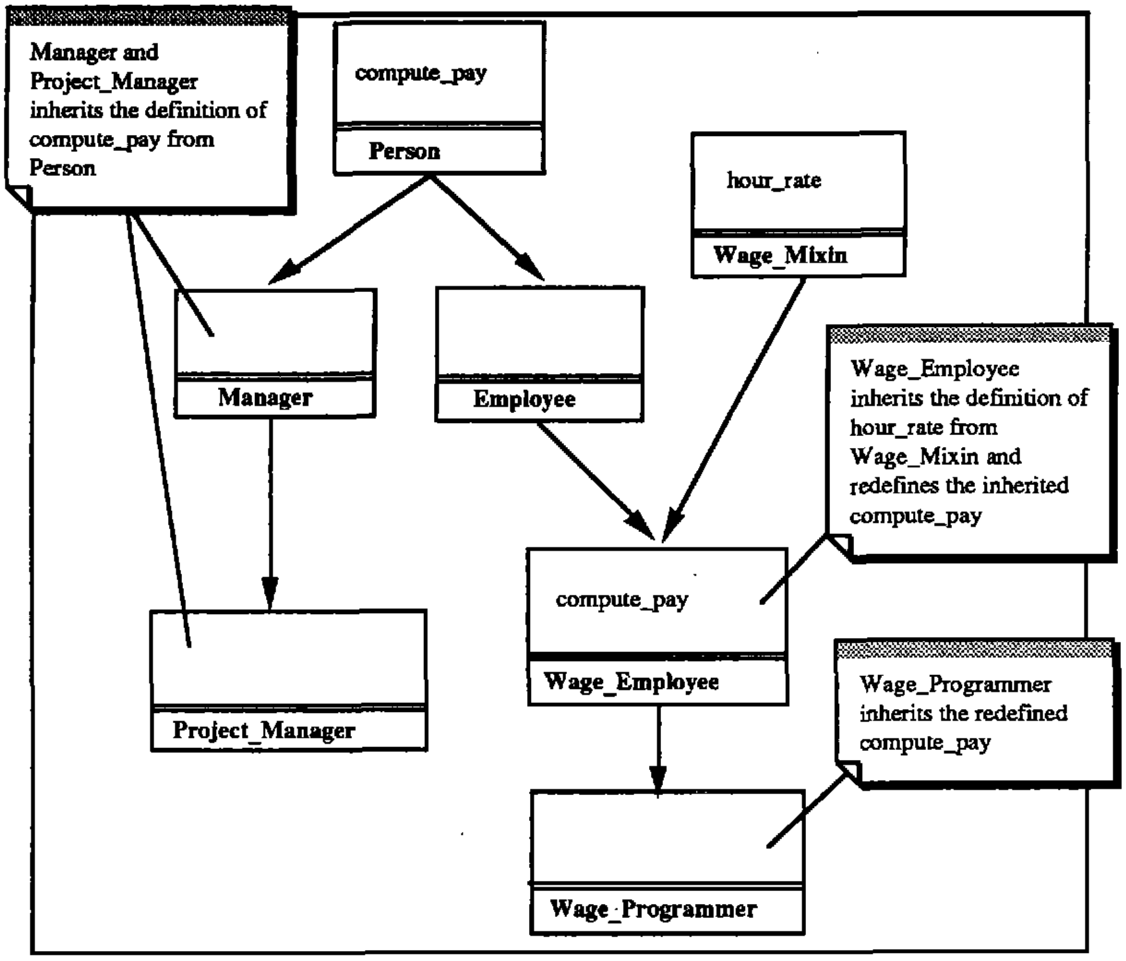

Manager and Project_Manager inherits the definition of compute_pay from Person
compute_pay
Person
hour_rate
Wage_Mixin
Manager
Employee
Wage_Employee inherits the definition of hour_rate from Wage_Mixin and redefines the inherited compute_pay
compute_pay
Wage_Employee
Project_Manager
Wage_Programmer inherits the redefined compute_pay
Wage_Programmer

Figure 3-2. Representation for Class Inheritance in IDEF4.

Figure 3-2 illustrates the representation in IDEF4 family for modeling a class-inheritance hierarchy. The arrows, in the illustration, point from superclasses to subclasses. In the figure, *Manager* is a subclass of the class *Person*; indicating that any instance of *Manager* is also a specialization of an instance of *Person.* Furthermore, any behavior exhibited by a person will also be exhibited by an instance of *Manager* unless the behavior is specialized or redefined in the definition of the *Manager* class. In the example, the class *Employee* is a direct subclass of *Person* and the class *Wage_Employee* is an indirect subclass of class *Person.* Each subclass inherits the characteristics (features) associated with its direct and indirect superclass(es). For example, the class *Wage_Employee* inherits the features from both the *Wage_Mixin* and *Employee* classes and redefines the feature *compute_pay.*

In the inheritance hierarchy, features reappearing in the subclasses indicate that those features are redefined (additional constraints or a new definition) in the subclasses. The *compute_pay* feature is first presented in class *Person*, but redefined in class *Wage_Employee.*

From the module point of view, inheritance is a macro-like "virtual copy" operation: all features associated with a superclass are automatically inherited by its subclasses, with the exception of those features that are redefined in the subclass. For example, in Figure 3-2, the *Person* class defines a feature named *compute_pay.* This feature will be inherited in all of its subclasses: *Manager*, *Project_Manager*, *Employee*, *Wage_Employee*, and *Wage_Programmer.* The definition for *compute_pay* in *Manager*, *Project_Manager*, and *Employee* is identical to that in *Person.* However, because *compute_pay* reappears in *Wage_Employee*, it is said to be "redefined" for that class and its subclasses. Since *Wage_Programmer* is a subclass of *Wage_Employee*, the definition applied to the *compute_pay* in *Wage_Employee* will be the definition for the *compute_pay* in *Wage_Programmer.*

A subclass is able to inherit (copy and use) any feature of its superclasses, but not vice versa. The notion of inheritance conflicts with the traditional notion of encapsulation (information-hiding) (Snyder 86). This violation, because it is allowed in a controlled way (and in one direction only), is one of the keys to the power of the object-oriented paradigm. A properly structured OOD uses inheritance facilities to minimize duplication of modules. The IDEF4 method is focused on structuring both classes and methods into two inheritance hierarchies - class inheritances and method taxonomies to ensure that the resulting designs have no duplication.

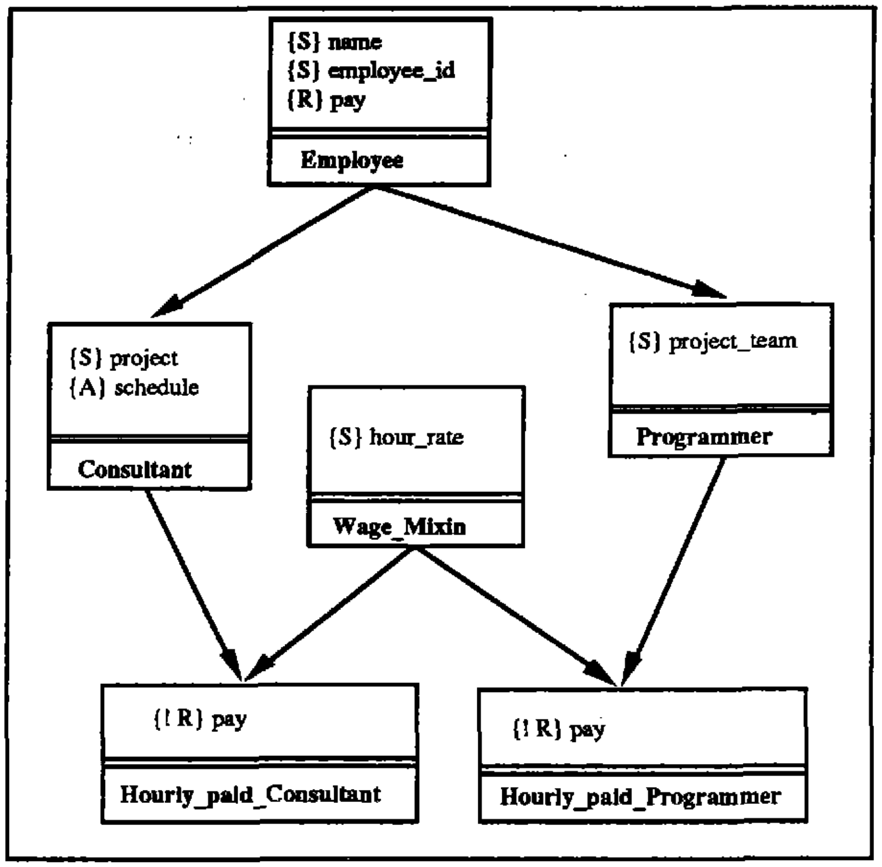


Figure 3-3. Partial Inheritance Diagram for an Employee Class.

In the IDEF4 method family, the class inheritance relationships are represented in the inheritance diagram as shown in Figure 3-3. An inheritance diagram provides information that describes the classes, their features, and any redefinition of features. For example, the reader familiar with IDEF4 syntax can determine that an instance of *Hourly_paid_Programmer* inherits all features of the classes *Employee*, *Programmer*, and *Wage_Mixin*. Furthermore, it can be seen that the feature *pay* is a routine that has been redefined in both *Hourly_paid_Programmer* and *Hourly_paid_Consultant*. The details of inheritance diagrams will be discussed in Section 4.3.

3.3 Features

"Feature" is a generic term used to capture either the state or behavior of instances of a class. In the IDEF4 method concept, a feature may be value-returning or side-effecting. For example, the class *Employee* has a feature *salary* that returns (value-returning) the salary of an employee, and a feature *print_paycheck* that prints out the employee's paycheck (side-effecting). However, whether a given value-returning feature is implemented by storage or by computation is functionally irrelevant in the initial design. That is, whether *salary* is implemented as a storage (a variable) or whether the value is computed from other features of the employee (a function) is not necessarily of concern in the initial design stages. This capability of the delay of decision-making is supported in IDEF4 methods by the hierarchy of feature taxonomy shown in Figure 3-4, which is presented by using the class diagram syntax.

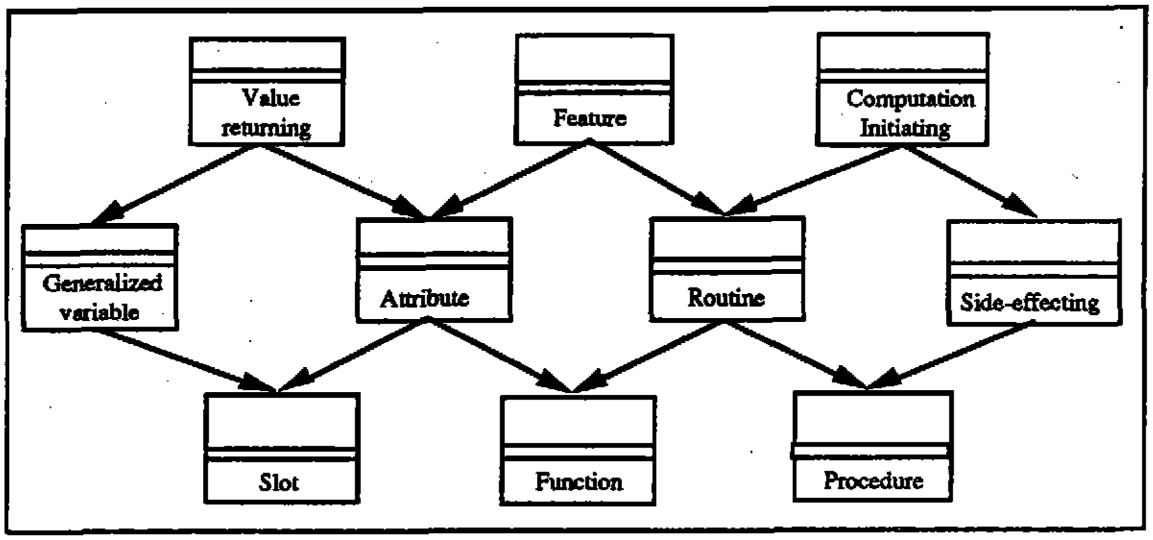


Figure 3-4. Feature Taxonomy Hierarchy.
Reprinted from (Mayer 92a).

3.3.1 Taxonomy of Features

The feature taxonomy allows features to be characterized in more general representation initially; then, gradually, to be defined more specifically as the design evolves. For example, a designer might first specify a characteristic of a class as a feature. Then, as the design evolves, the designer can specialize the definition of that feature to an attribute, a routine, a slot, a function, or a procedure as shown in Figure 3-4.

Attributes represent those features that return values when queried (value-returning), whereas routines represent the features which, when appropriately triggered, will initiate a computational operation. Note that attributes and routines are not mutually exclusive. Along the evolution of the design process, attributes can be refined into slots or functions, and routines can be refined into functions or procedures. Slots are those features that are characterized as storage-type variables. Functions are features that are both value-returning and computation-initiating; they return a value by computing it whenever queried. Procedures are computational features that do not return any values; they are only executed for their side-effects.

3.3.2 Inheritance of Features

All features defined in the superclass are automatically inherited by the subclass through the inheritance of classes. Figure 3-5, for example, shows that the class *Employee* has a feature *pay*. Subclasses of *Employee* such as *Wage_Employee* and *Salary_Employee* inherit the *pay* feature from *Employee*.

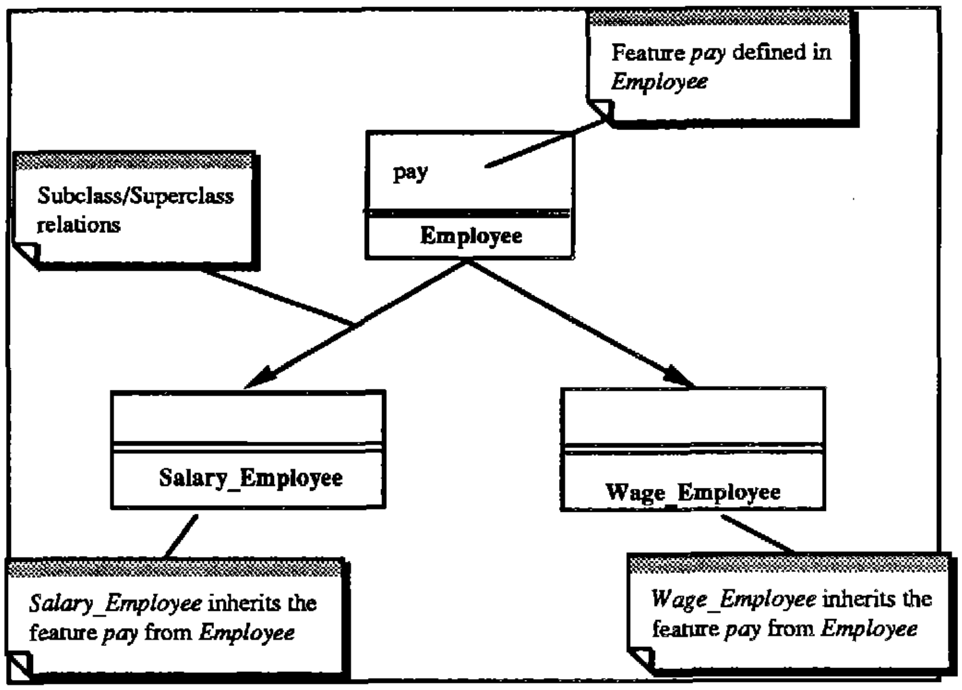

Feature *pay* defined in *Employee*
pay
Employee
Subclass/Superclass relations
Salary_Employee
Wage_Employee
Salary_Employee inherits the feature *pay* from *Employee*
Wage_Employee inherits the feature *pay* from *Employee*

Figure 3-5. Inheritance of Features.

If *pay* is implemented as a computational feature, then using the general payment calculation of employee for a wage employee would be inefficient. Therefore, it might be desirable to redefine the *pay* for *Wage_Employee* that it will use the more specialized calculation for those wage employees. This specialized calculation would be invoked instead of the more general *pay*. Thus, the more specialized feature "shadows" (redefines) the more general feature. Figure 3-6 illustrates a case in which feature *pay* in *Wage_Employee* shadows (redefines) the generic *pay* in *Employee*. The *Salary_Employee* class continues to inherit *pay* from *Employee*.

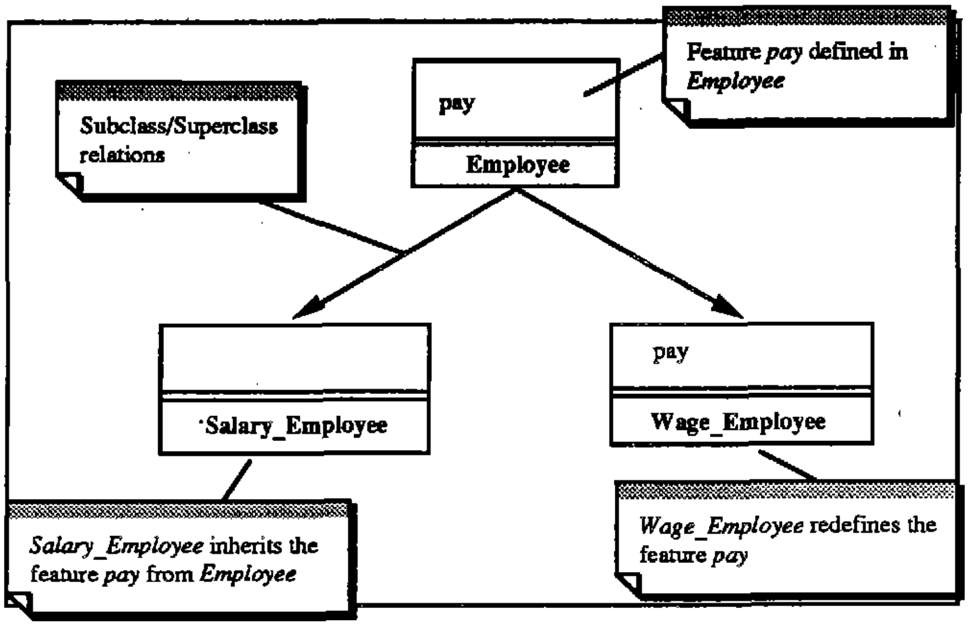

Feature *pay* defined in *Employee*
pay
Employee
Subclass/Superclass relations
Salary_Employee
pay
Wage_Employee
Salary_Employee inherits the feature *pay* from *Employee*
Wage_Employee redefines the feature *pay*

Figure 3-6. Redefining Inherited Features in Class Inheritance.

3.3.3 Presence of Features

The presence of a feature indicates the way the designer intends to associate that feature with a class (i.e., defined in the class, redefined in the class, or an inherited feature). Figure 3-7 shows the classification for the kinds of "feature presence" provided by IDEF4.

Using this classification scheme, a feature that is associated in any way with a class is said to be present in the class. Those features whose names are displayed in the class box of a class *A* are said to be directly present in *A*. Those features present in a superclass *B* of *A* are considered to be present in *A* as well, and are actually inherited features of *A*. Features of *A* that are both directly present and inherited in *A* are redefined in *A*. They are directly present because the class *A* is giving additional or revised information (constraints) about them that is not present in the superclasses of *A*. Features that are directly present but not inherited in A are said to be defined in *A*; those that are inherited but not directly present are said to be virtual in *A*[9].

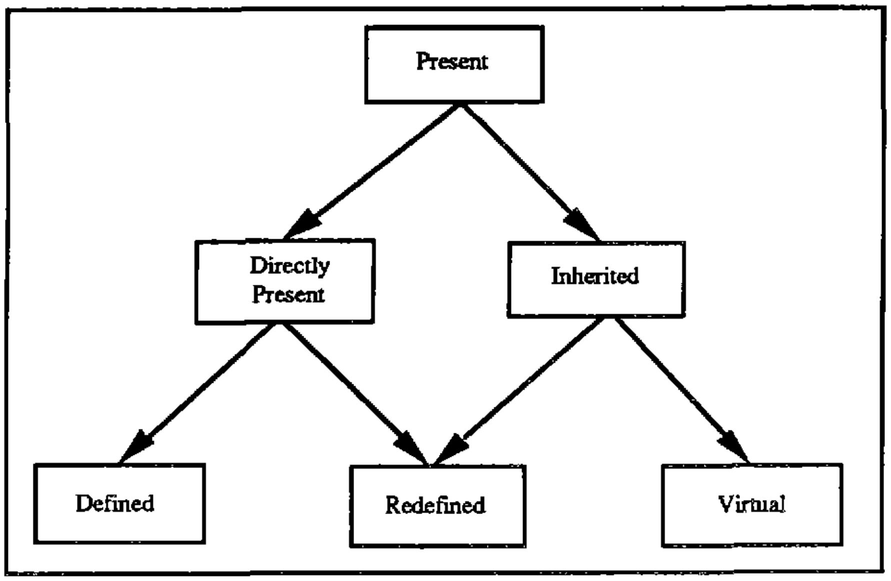


Figure 3-7. Classification of the Presence of Features.

For example, Table 3-1 categorizes the *pay* feature of the class hierarchy described in Figure 3-6. Table 3-1 shows the classification for the *pay* feature with respect to each class in the hierarchy as 1) present, directly present, and defined in the *Employee* class; 2) present, inherited, and virtual in the *Salary_Employee* class; and 3) present, directly present, inherited, and redefined in the *Wage_Employee* class.

[9] The term 'virtual' addressed here is different from the ones used in C++. In C++, 'virtual function' is used to enforce the run-time binding mechanism, and 'virtual' inheritance is used to resolve the name conflicts occurred in a multiple inheritance.

Class	Present	Directly Present	Inherited	Defined	Redefined	Virtual
Employee	X	X		X		
Salary_ Employee	X		X			X
Wage_ Employee	X	X	X		X	

Table 3-1. Presence of the Feature Pay in the Employee Class Hierarchy.

3.3.4 Type of Features

In the IDEF4 method family, value-returning features have a return type defining the type of their return value. The return type can be a primitive type such as integer or character supported by an implementation language; a class that defined elsewhere in the design; or a collection of other classes. From the design management point of view, the return type of a feature provides a means of expressing other associations between classes. These associations between classes are not visible in the class inheritance lattice presented in the inheritance diagrams. In the IDEF4 method family, they are captured in the type diagrams. Experience has shown that these associations are as important as inheritance relationships among classes. The management of these associations is critical to the development of large object-oriented systems. Only through the careful study and design of types of features, can the development team capture the intended domain relations and evolve the design in an orderly fashion.

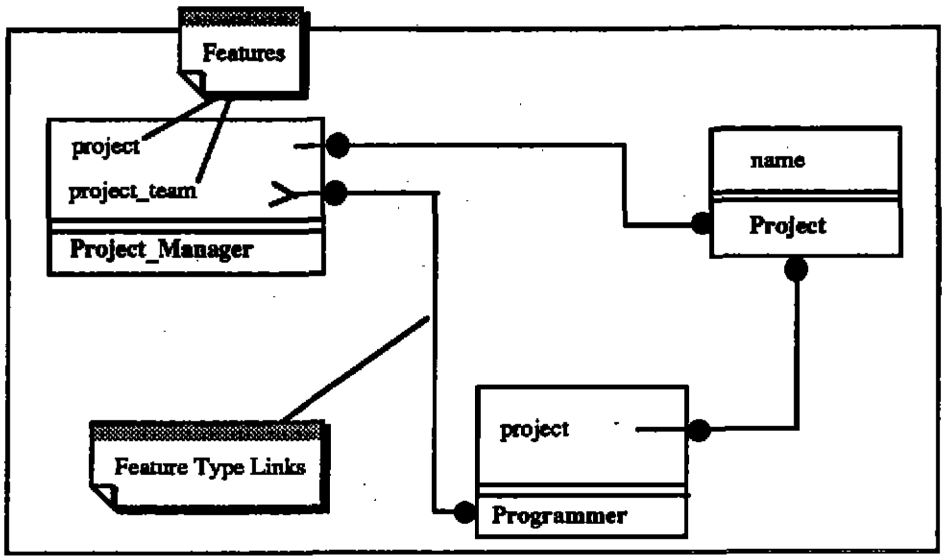


Figure 3-8. A Partial Type Diagram Defining Types of Features.

Figure 3-8 illustrates an example of a type diagram that defines the feature-return-type relations. The *Project_Manager* class defines two features (*project* and *project_team*). Feature *project* returns an object of class *Project*, specifying the current project that the project manager is conducting. Feature *project_team* returns a set of objects of class *Programmer*, specifying the members in the project team under that project manager. The class *Programmer* also defines a feature *project* which specifies the current project that the programmer is working on.

3.4 Methods

As discussed previously, a class may have features that define the behaviors of its object instances. The features that define behaviors are computation-initiating, and by definition, they can be routines, functions, or procedures. Accordingly, the functions and procedures are specializations of routines (see Figure 3-4). These computation-initiating features are listed in groups according to the generic

behaviors they specify in CIDSs. For example, consider a *Drawable_Object* class that might have a generic behavior, *draw*, which is specified in its CIDS. For all the drawing behaviors possessed by the instances of the *Drawable_Object* class, *draw_to_screen* and *draw_to_printer*, they will be included under the generic *draw* behavior. However, the feature kinds (routines, functions, or procedures) of *draw_to_screen* and *draw_to_printer* might change over the evolution of the design.

3.4.1 Methods, Contracts, and Method Sets

In OOPLs, each computation-initiating feature is implemented by a single method That method provides the required computation for the behavior specified by the feature. However, the notion of a method in IDEF4 is not the same as the usual notion of a method from an object-oriented language point of view. In object-oriented programming, a method is an executable piece of code which algorithmically specifies the computation to be performed by means of a set of language statements. For example, a C++ member function. In IDEF4, on the other hand, **methods** are defined by the **contract** that they must fulfill. In fact, IDEF4 does not specify an individual method; rather, **method sets**. Any of the methods in the set can fulfill a specific contract. In other words, we refer to the contract for a method rather than the code; this is based on the notion of S-correct of design, which was discussed in Section 2.1. The contract for the associated method set is documented in the Contract Data Sheet (CDS) related to that method set.

Computation-initiating features specify the behaviors of the instances of the class. In IDEF4, each feature is mapped through the dispatch mapping to a method set in the method submodel. This method set documents the constraints (in the associated CDS) for implementing these behaviors. Some computation-initiating features may be presented in more than one class and may intend to specify different specific behaviors depending on the class of objects upon which they perform their computations. In other words, a generic behavior may have different method sets mapped in different classes.

For example, the routine *compute_raise* is a computation-initiating feature defined and redefined in class *Employee* and *Manager* respectively (where *Manager* is a subclass of *Employee*). It is also included under the generic behavior *raise* which is specified in both the CIDSs of *Employee* and *Manager*. In a programming language, different method implementations would be defined for each of these routines, such as *get-raised-as-employee* and *get-raised-as-manager*. In IDEF4, on the other hand, individual methods are not represented. *get-raised-as-employee* and *get-raised-as-manager* would refer to method sets and their related contracts, which will be illustrated in the *raise* method taxonomy (to be discussed in the following section). Therefore, the computation-initiating feature *compute_raise* and its class *Employee* together specify a method set *get-raised-as-employee*. Any method in the method set *get-raised-as-employee* would satisfactorily implement the feature *compute_raise* for the class *Employee*. The idea in IDEF4 is to describe or design the behavior, not program the behavior.

3.4.2 Taxonomy of Method Sets

A method set can be considered as a computational characteristic defined by a set of constraints that will pick out a set of possible correct implementations. This set of constraints is called the contract for the method set. In an IDEF4 design, the concern is with the definition of the contract, rather than individual method implementations in the set.

However, similar contracts can be grouped together according to the behaviors specified by the computation-initiating features that the method sets associate with. In IDEF4, a Method Taxonomy Diagram classifies a generic behavior into several method sets according to the similarity of their contracts. For example, the method taxonomy diagram in Figure 3-9 illustrates the design of the method sets for the generic *Print* behavior. Each box in the graph represents a method set, which requires a contract constraining the implementation of the methods in the set. The method sets in a method taxonomy diagram are arranged in a more-specific order from left to right or from top to bottom; re-definition or additional constraints might be added to the contracts of those more specific method sets. For instance, in the *Print* method taxonomy diagram, the least-specific method would be *Print-object*. However, for some classes such as *Text-screen-object*, additional constraints would be required for the method set *Text-screen-print* which maps to the *Print* routine of *Text-screen-object* as shown in the figure. The dispatch mapping between a method set and a routine (a computation-initiating feature) is specified by "[ ]". The additional symbol "!" on the left of the *Print* routine indicates that the routine has been entirely redefined. For a redefined routine, the contract for the method set dispatch-mapped with the routine may override or conflict with the preceding method set contracts. More details about additional symbols are discussed in Section 4.3.4.

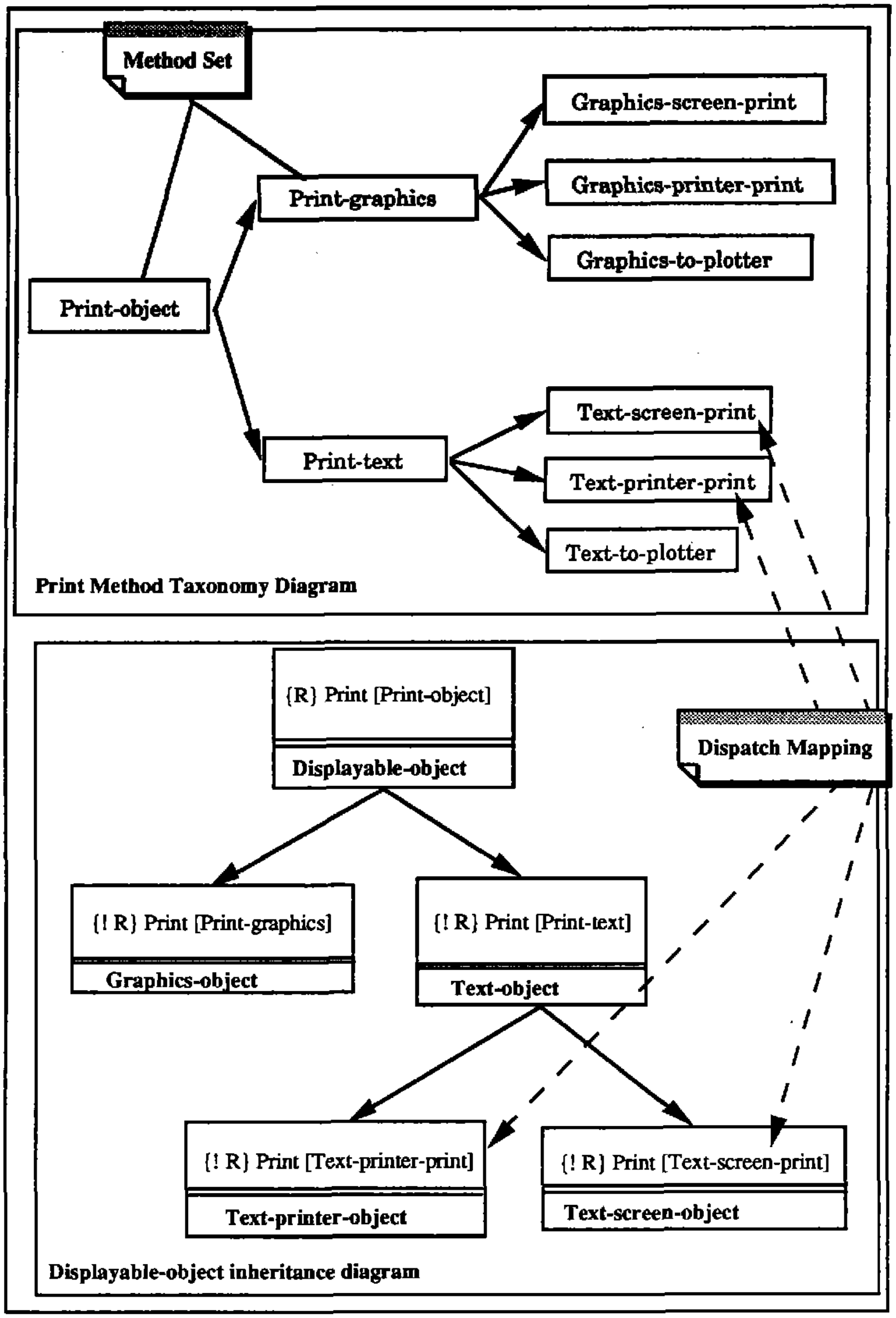


Figure 3-9. Print Method Taxonomy and Associated Routines.

3.5 Constraints

In IDEF4 methods, constraints are used for specifying both class-invariant definitions and contracts on method sets. Constraints will often be specified in natural language statements in the design evolution. As the design progresses, constraints will be refined and specified more formally (i.e., in a formal language such as first-order predicate logic).

For example, the class-invariant constraint on the feature *Identity-number* of type *integer* in the class *Employee* may be expressed as:

"The *Identity-number* feature in the *Employee* class must be a unique *integer* over all instances of the class *Employee*."

More formally, this might be written as the following constraint specifying that no two employees may have the same identity number:

For-all(x y) (employee x)^(employee y)^(not-equal x y)

^(not-equal(identity-number x)(identity-number y)).

These constraints are hold as relations among design elements (classes, features, and methods) that must be enforced by the system (implementation / program). Class-invariant constraints are documented in class invariant data sheets (CIDSs) associated with class boxes in inheritance diagrams. Method set contracts are documented in contract data sheets (CDSs) associated with method sets in method taxonomy diagrams. As the design evolves, these constraints will be refined more specific to be implemented; in IDEF4/C++ implementation design, CIDSs will be the major design specifications for the C++ class definitions, and CDSs will be the specifications for function implementations.

CHAPTER IV

METHOD SYNTAX

4.1 Organization of IDEF4/C++ Diagrams

In this chapter, we present the IDEF4/C++ notations and its syntactical elements. The organization of the extended notations will be described in the first place. Discussion of the each diagram will contain a concise description of their graphical elements and the examples that demonstrate the use of the diagrams.

A completed IDEF4/C++ model consists of a class submodel and a method submodel. Each submodel has diagrams and data sheets as model components. The class submodel provides a system state view for the design, whereas the method submodel provides a system behavior view. These submodels are connected through dispatch mapping, as introduced in Figure 3-9, which is specified in both inheritance diagrams and method taxonomy diagrams while Figure 4-1 gives an overall picture of the organization of IDEF4/C++ diagrams. As shown in Figure 4-1, diagrams are grouped into two submodels as follows. Each diagram type presents a unique perspective and provides a mechanism for viewing and devising the design.

- Class Submodel
    - Class Lattice Diagrams
      Class lattice diagrams provide a view for browsing the class lattice.
    - Inheritance Diagrams

Inheritance diagrams describe inheritance relationships and those directly presented features in the class boxes.

- Type Diagrams
  Type diagrams specify return types of features or compositional relationships among classes.

- Friend Diagrams
  Friend diagrams declare the C++ friend associations between a class and its friend functions and classes.

- Template Diagrams
  Template diagrams specify C++ class template declarations.

- Instantiation Diagrams
  Instantiation diagrams validate the design by giving existing composite relationships between instances.

- Method Submodel

  - Method Taxonomy Diagrams
    Method taxonomy diagrams classify method sets by their behavioral similarity.

  - Client Diagrams
    Client diagrams specify the calling relationships between functions or procedures.

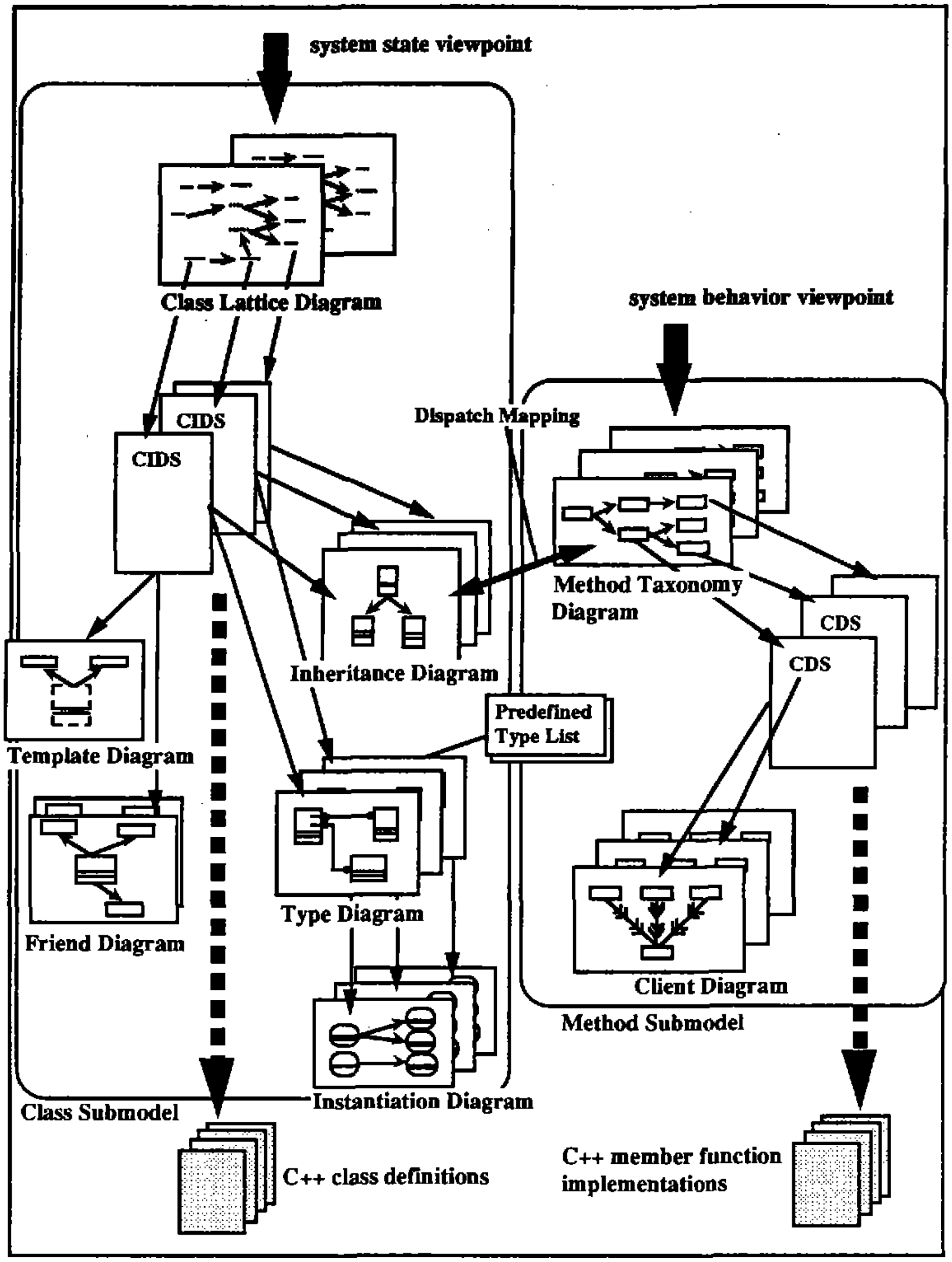
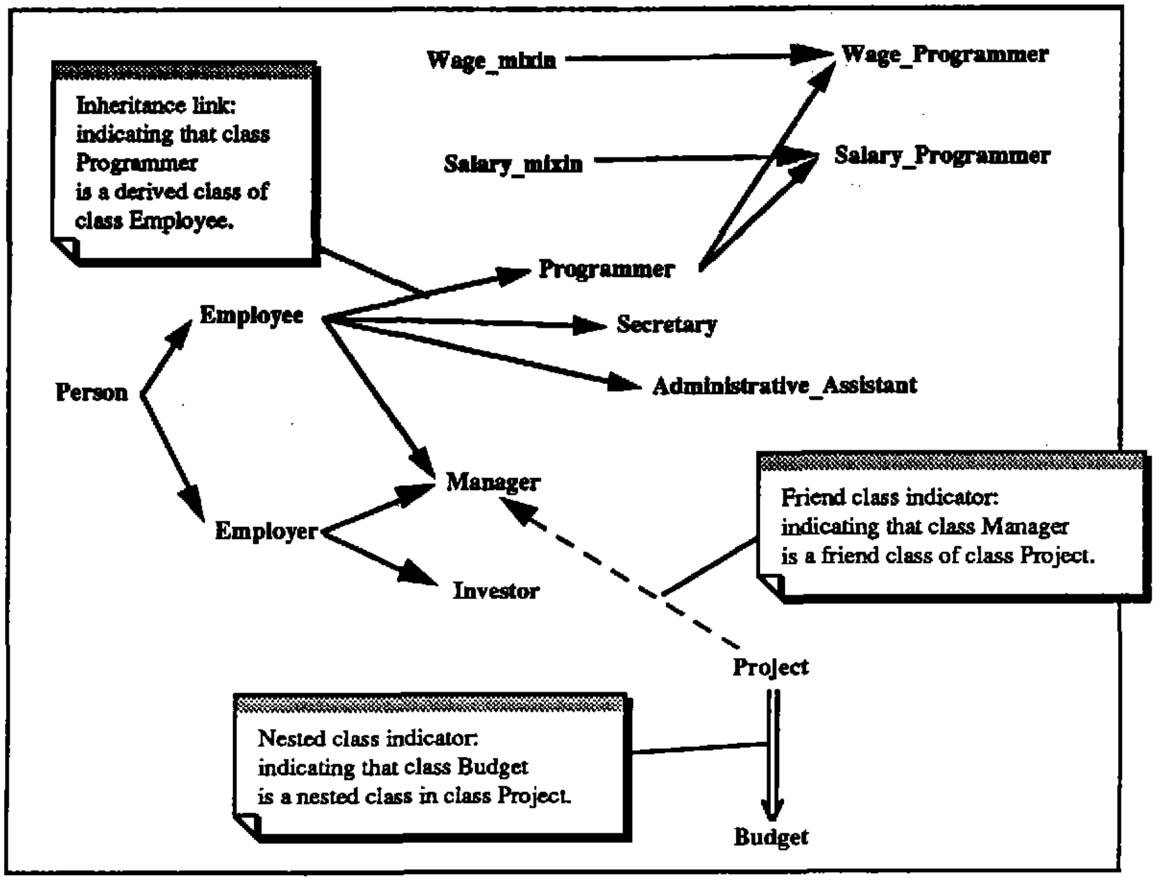


Figure 4-1. Organization of IDEF4/C++ Diagrams.

As a language-specialized design method, IDEF4/C++ adds the extensions to the generic IDEF4. Feature symbols are extended to be used for the C++ class member declarations. The user predefined data type list is provided as a supplementary device for type diagrams for collecting those user predefined types in the design. Class invariant data sheets (CIDSs) and Contract data sheets (CDSs) are also extended to be able to provide more specific information for coding C++ class definitions (usually managed in the .h/.hpp files) and member functions (.cpp files), respectively.

4.2 Class Lattice Diagrams

Class lattice diagrams are used to illustrate the class lattices, which browse the whole class submodel or a particular part of the submodel. To provide a top abstract view for the class submodel, only class names are shown in the diagram. Three class relationships are also described in the lattice, they are graphically presented in terms of arrows. Figure 4-2 illustrates an example class lattice diagram. As a language extension of IDEF4, friend class and nested class links are included into the presentation. Inheritance link arrows (shown as normal arrows) point from the base classes to the derived classes.

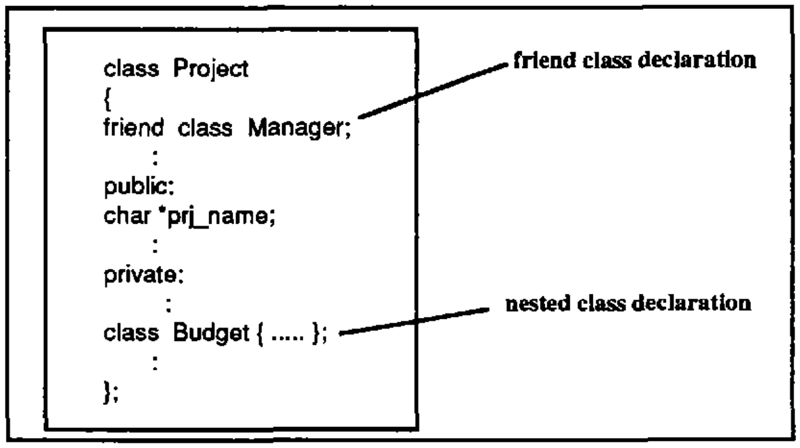


Wage_mixin
Wage_Programmer
Salary_mixin
Salary_Programmer
Inheritance link: indicating that class Programmer is a derived class of class Employee.
Programmer
Employee
Secretary
Administrative_Assistant
Person
Manager
Employer
Investor
Friend class indicator: indicating that class Manager is a friend class of class Project.
Project
Nested class indicator: indicating that class Budget is a nested class in class Project.
Budget

Figure 4-2.

Example Class Lattice Diagram for an Employee Management System.
*Arrows with a dotted line point from a class to its friend classes.
*Arrows with a double line point from the defining classes to their nested classes.

As shown in Figure 4-2, class *Person* is the root class of the lattice, where class *Employee* and *Employer* are its directly derived classes. Three kinds of employees defined are *Programmer*, *Secretary* and *Administrative_Assistant*. Inheritance link arrows pointing from *Employee* to each of these classes indicates that they all are directly derived classes of class *Employee*. Two classes are designed as mixins for work pay (salary/wage) calculation purposes; *Wage_mixin* and *Salary_mixin*. By using these mixins, different work pay types of employees can be derived, such as class *Wage_Programmer* and *Salary_Programmer*. Note that there is a nested class indicator arrow pointing from class *Project* to class *Budget*, which indicates that the class *Budget* is declared within the scope of the definition of the class *Project*. The design intention here is to make the class *Budget* invisible from the rest of the system, excluding the class *Project*. This prevents the information kept in *Budget* from being accessed by other classes accidentally or on purpose. The C++ code example for class *Project* is shown in Figure 4-3. We declare *Budget* as a private member of *Project* to prevent any explicit access to the budget information[10]. A friend indicator, pointing from *Project* to *Manager*, indicates that the class *Manager* is declared as a friend class of *Project*. By doing so, *Manager* is able to access any features of *Project*, such as the budget information. By using a friend class declaration (*Manager*) and a nested class declaration (*Budget*) together, we provide a more safe mechanism for accessing the class *Project* and its budget information. However, the class lattice diagram provides a means for presenting the design intention such as the one discussed above, which is very handy and important especially when we deal with large-scale systems that contain massive numbers of classes.



```
class Project
{
friend class Manager;
    :
public:
char *prj_name;
    :
private:
    :
class Budget { ..... };
    :
};
```

Figure 4-3. Example C++ Code for the Class Project.

[10] If the nested class *Budget* is declared as a public member of *Project*, it can be explicitly accessed by using the class identifier - Project::Budget.

4.3 Class Inheritance Diagrams

As mentioned in Chapter III, inheritance diagrams are used to describe inheritance relationships between classes. Extensions for the inheritance diagrams include feature access control, types of inheritance links, and feature symbols. The basic syntax consists of the class box, symbols that describe features, and arrows presenting the inheritance links.

4.3.1 Extended Class Box Syntax

An IDEF4/C++ class box groups features into three areas: public, protected, and private, for describing feature access. Public features are those that appear in the top group and are visible to (accessible by) the rest of the system. Protected features are those that appear in the second group and are accessible by the owner class and its **directly** derived classes. Private features appear in the third group. They can not be accessed by any other class except for the owner class. Recall that, in IDEF4, there are only two types of feature access control: public and private. The private features in IDEF4 can be accessed by their owner class as well as **all** its derived classes. This is different from IDEF4/C++ and should be noted when evolving a generic IDEF4 design into a IDEF4/C++ implementation design. Generally, the default translation is to transform these generic IDEF4 private features to be the protected features in IDEF4/C++. Figure 4-4 presents the class box syntax and an example class box for the class *Project*. Where features *project_name*, *project_no*, *project_manager*, and *project_team* are public features; *Budget* is a protected feature and *internal_id* is a private feature.

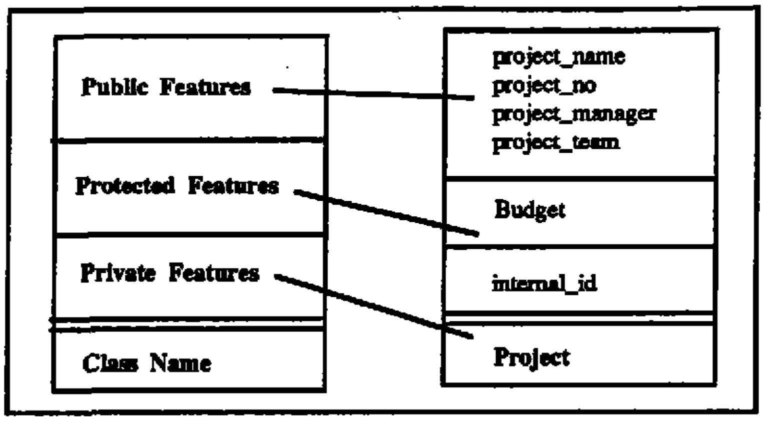


Figure 4-4. Extended Class Box Syntax.

An inheritance relationship (link) between classes is presented by an arrow pointing from a base class to its derived class in the class inheritance diagram. For example, following the class lattice described in Figure 4-2, there is an inheritance relationship between the class *Person* and the class *Employee*: *Employee* is derived from *Person*. The inheritance relationship is illustrated in Figure 4-5.

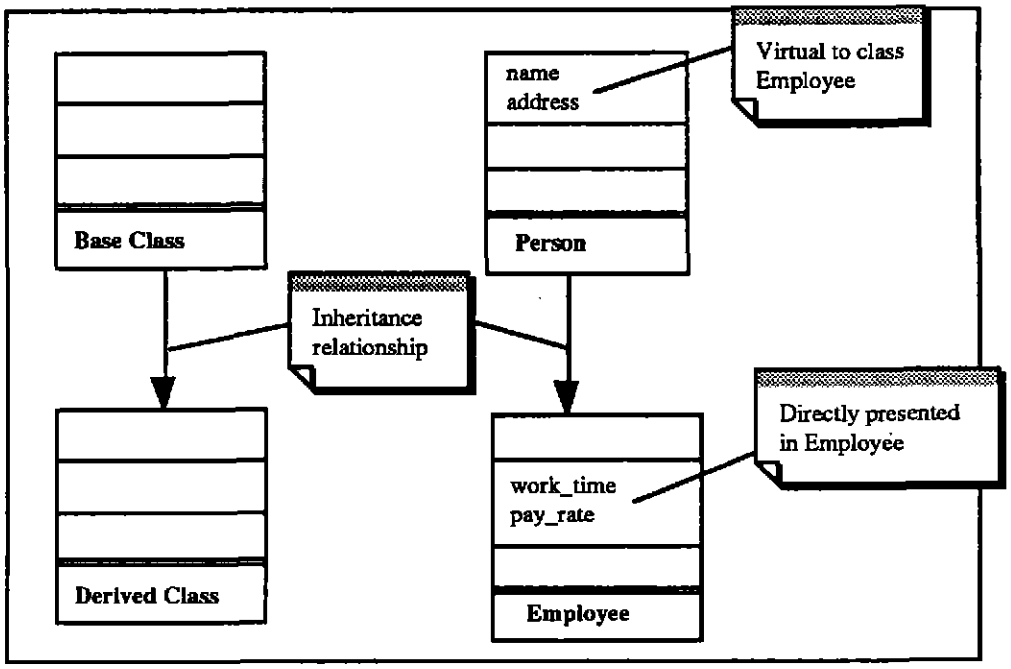


Figure 4-5. Inheritance of a Base Class and a Derived Class.

4.3.2 Public, Protected, and Private Inheritance Links

In IDEF4, the feature access types of inherited features in a subclass are the same as they were in the base class. In IDEF4/C++, access types of inherited features in a derived class are determined by both the original access type and the type of inheritance links. IDEF4/C++ supports three types of inheritance links: public, protected, and private. Different types of inheritance links give different effects on determining the access types of inherited features. Non-private features (public/protected) of a base class become private features of its derived class in a private inheritance link. Non-private features of a base class become protected features of its derived class in a protected inheritance link. Non-private features of a base class will keep their original access in the derived class in a public inheritance link . Figure 4-6 compares three different types of inheritance links between class *Person* and class *Employee*, where *Person* has a public feature *name*, a protected feature *SSN* and a private feature *internal_id*. The feature *internal_id* will not be presented in *Employee* since it is private to *Person*. As shown in the figure, *name* will remain public and *SSN* will remain protected in *Employee*, if there is a public inheritance between *Person* and *Employee*. For a protected inheritance, both *name* and *SSN* will become protected features of *Employee*, but for a private inheritance, both *name* and *SSN* will become private.

IDEF4 has no provision for different types of inheritance links. Yet, in IDEF4, since all inherited features will keep the same access (public to public, private to private) in a subclass, the type of inheritance is considered as public. Therefore, while evolving a generic IDEF4 design towards IDEF4/C++, if the type of an inheritance link has not been further specified, the inheritance type is public by default.

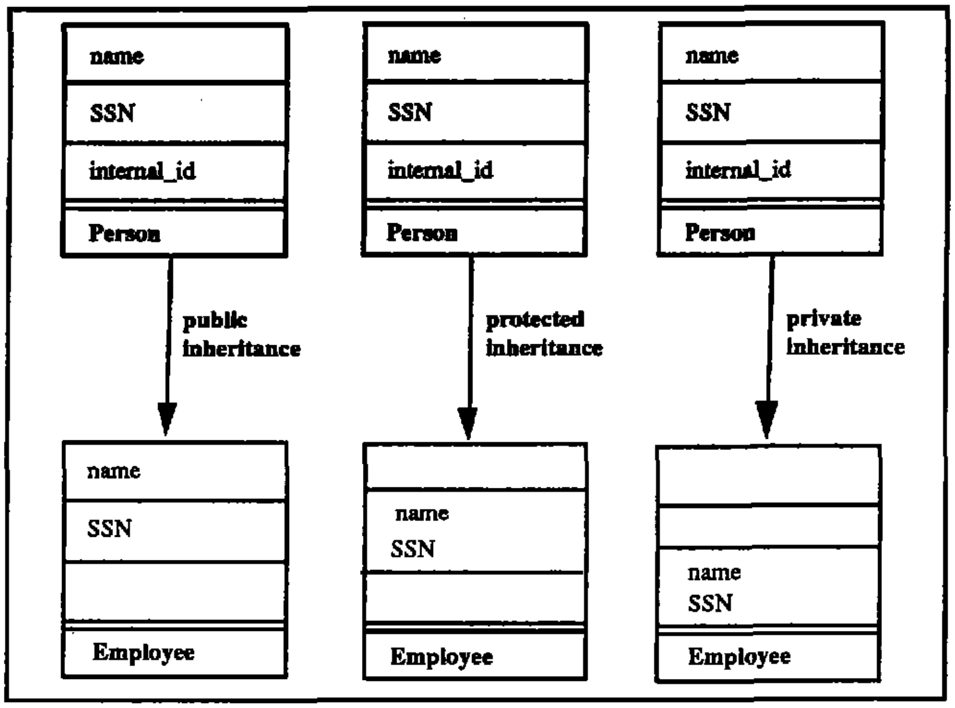


Figure 4-6. Comparison of Public, Protected, and Private Inheritances.

4.3.3 Virtual Inheritance

In the previous chapter, we discussed the inheritance of features finding that a derived class will inherit all the characteristics/constraints from a base class unless they have been redefined. However, the base classes, if there is more than one, must agree on the common characteristics/constraints among them. This is usually referred to as the name conflict problem in multiple inheritance as described in Section 1.2. In IDEF4/C++, name conflict is resolved by specifying the inheritance as *virtual*.

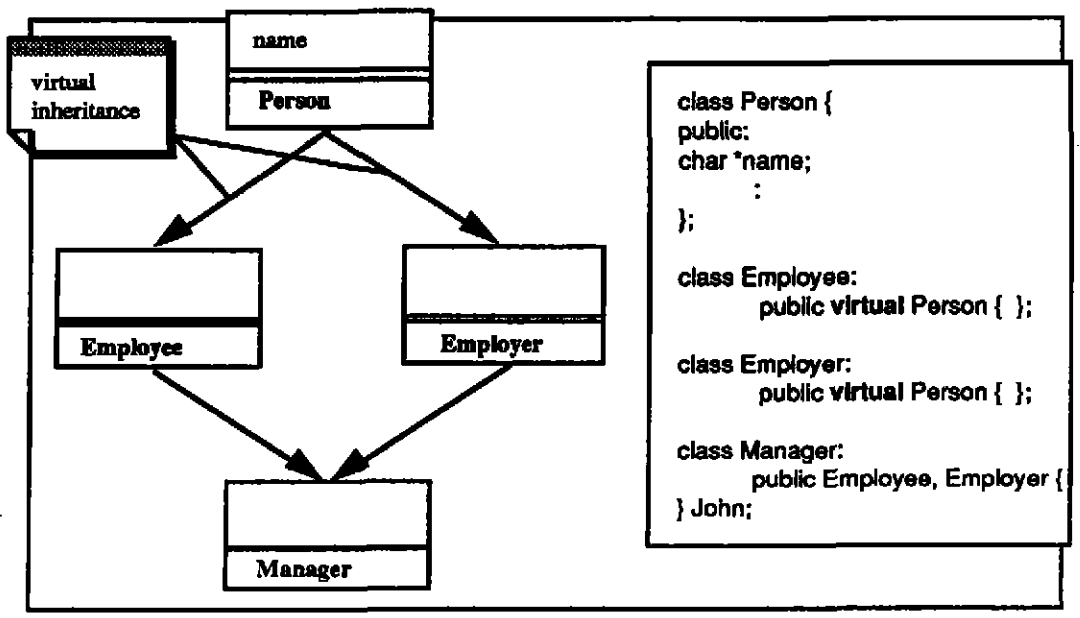
For example, consider the *Manager* class illustrated in Figure 4-2. *Manager* is a derived class from both *Employee* and *Employer*, which are all derived from the class *Person*. If *Person* has a non-private feature *name*, both *Employee* and *Employer* will inherit the *name* feature. If the inheritance links between *Manager* and *Employee*, *Manager* and *Employer* are not specified as virtual, the class *Manager* will keep both copies of *name*: one from *Employee* and the one from *Employer*. To access the *name* features in every instances of *Manager*, one has to explicitly specify the class identifiers:

```
John.Employee::name
John.Employer::name
```

This is certainly awkward. To be more elegant, one would like to specify both inheritance links between *Employee* and *Manager*, *Employer* and *Manager* to be "virtual". The inheritance diagram and its associated C++ code are illustrated in Figure 4-7. Thus, the access to the feature *name* in each instance of *Manager* can be quite straight-forward by using the following statement:

```
John.name
```

In summary, IDEF4/C++ provides a virtual inheritance declaration for resolving name conflict in multiple inheritance and public, protected, and private inheritance link types for managing access control of inherited features. Note that all these extensions are to be specified in the CIDS associated with the focused class. Refer to Section 4.3.6 for the format of CIDSs.



```
class Person {
public:
char *name;
        :
};

class Employee:
        public virtual Person { };

class Employer:
        public virtual Person { };

class Manager:
        public Employee, Employer {
} John;
```

Figure 4-7. Virtual Inheritance in IDEF4/C++.

4.3.4 Feature Symbols and Their Extensions

Recall that we have presented the IDEF4 feature taxonomy in Figure 3-4, which demonstrates that features may be specified as routines, attributes, slots, functions, or procedures as the design evolves. This delayed-decision practice in the design process is perfectly acceptable and is likely to be the case in the early definition of the features in a generic design. As the development of the design continues, the designer will classify these features. Feature symbols are used to represent the classification of features. For example, '**A**' represents attributes. '**R**' represents routines. '**S**' represents slots. '**F**' represents functions and '**P**' represents procedures. When a designer wishes to commit to the classification of a feature, the proper symbol may be added to the left of the feature name in the class box within braces '{ }'. Figure 4-8 shows an example of using these feature symbols;

indicating the ***name*** , ***department*** and ***internal_id*** features to be implemented as slots, *work_schedule* as an attribute, and *compute_pay* as a routine.

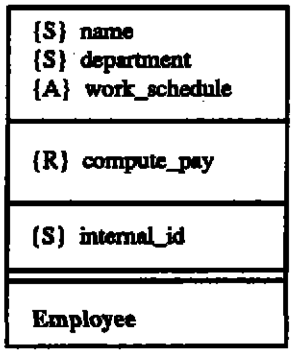


Figure 4-8. Example of Feature Symbols.

As discussed in Section 3.3.3, inherited features may be redefined in the derived class. Additional constraints or even a whole new contract may be given to redefine the feature. Two symbols are used to represent the redefinitions: '+' indicates that additional constraints are added to the contract associated with the named feature, '!' indicates that the named feature represents a new contract and has a same name as the inherited feature. Conceptually, a redefined feature with new contract ('!') conflicts and shadows the inherited one, whereas the feature with additional constraints ('+') just specializes the inherited feature. In Section 5.6, issues of conflicting and non-conflicting features will be discussed.

For example, consider that the *Programmer* class (Figure 4-9) is a subclass of the *Employee* class, but the algorithm for calculating the work pay (feature *compute_pay*) for an employee might be too general for a programmer (the programmer might be paid by project). For this reason, the designer must specify a redefinition for the *compute_pay* of *Programmer.*

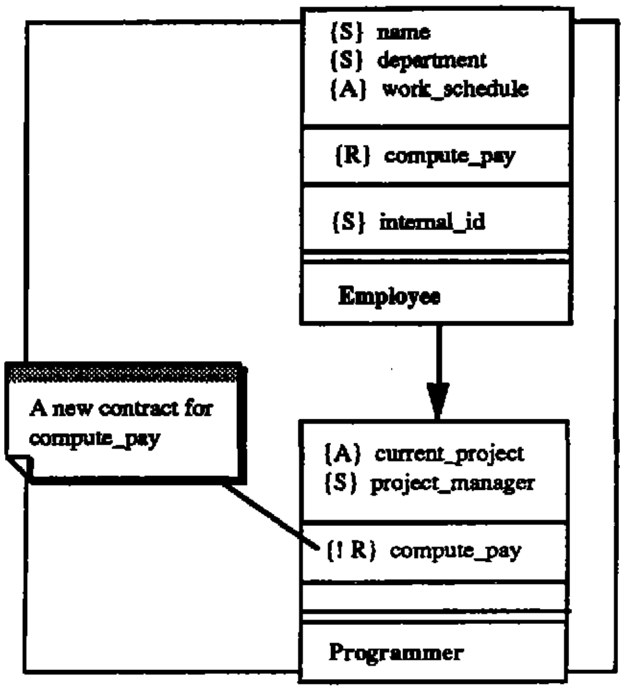


Figure 4-9. Example of a Redefined Feature with New Contract.

Generally, attributes and routines can be refined further into either slots, functions, or procedures. However, to devise an implementation design for C++ requires additional feature classification. In a C++ class definition, features are called class members. The data members are used to store the state of each object instance and the function members are used to implement the behaviors. Evolving features in an IDEF4 generic design towards IDEF4/C++ is quite straight-forward. Slots generally map to data members that are accompanied with their read/write functions. Functions map to member functions and procedures map to the member functions with *void* return types. However, there is more to be accomplished; C++ class members can be virtual, static, const etc. To be able to implement the design in C++, slots, functions, and procedures have to be further specified. IDEF4/C++ provides an additional set of symbols for this purpose. They are used along with the general symbols discussed previously and presented within the braces as well.

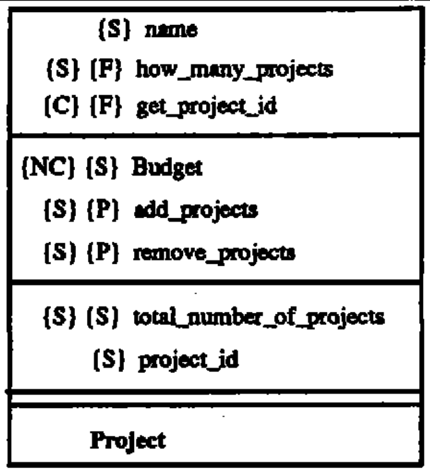
These additional symbols are described as follows:

VF	Virtual function - The 'VF' symbol indicates that the named feature is a C++ virtual member function. A member function declared as virtual in the base class can be overridden by the member function in the derived class that possesses the same name and signature but with different implementation.
V0	Pure virtual function - The 'V0' symbol indicates that the named member function is a C++ pure virtual function. A pure virtual function is declared only for inheritance purpose and since it serves as a placeholder, no implementation needs to be given. In other words, a C++ pure function is "pure" in the sense that it does not have a function body, only the function's signature is specified.
C	Const member function - The 'C' symbol indicates that the named member function is a const member function. A member function declared with *const* prevents any modification to the data members that the function accesses.
S	Static member - The 'S' symbol indicates that the named feature has only one copy among all the instances of the owner class. Data members declared to be static are therefore global in the owner class scope and do not need be replicated from one instance to another, thereby saving memory space and maintaining consistency. Similarly, static member functions are like global functions whose scope is within the owner class.

Usually, they are the member functions which access those static data members.

NC Nested class - The 'NC' symbol indicates that the named class is declared in the definition of the owner class as a data member. Without explicitly specifying the owner class's identifier, one can not access to the nested class.

Figure 4-10 in the following page gives an example of the presentation of these extended feature symbols and the related C++ code. Where *Project* has a nested class *Budget* and a static data member *total_number_of_projects*. *Project* also contains three static member functions; (1) *add_projects* ; (2) *remove_projects* for incrementing and decreasing the total number of projects; and (3) *how_many_projects* for querying the value. A const member function *get_contract_id* which guarantees access to the data without making any changes is also defined in *Project*.



```
class Project
{
 public :
 char *name;
 static int how_many_projects (void)
            { return total_number_of_projects; };
 int get_project_id (void) const { return project_id; };
            :
 protected :
 class Budget
        { public:
          int budget_number;
                     :
          };
 static void add_projects (void)
             { total_number_of_projects++;};
 static void remove_projects (void)
             { total_number_of_projects--;};
          :
 private :
 static int total_number_of_projects;
 int project_id;
         :
};
```

Figure 4-10. Example of Extended Feature Symbols.

4.3.5 Inheritance Diagrams

The inheritance graph of a particular IDEF4/C++ model will be simply a single, maximal inheritance diagram that shows all the classes and their direct inheritance relationships. An inheritance diagram of this size would have little practical use if actually drawn, but it is useful to keep in mind as a way of imagining the full scope of inheritance diagrams. In fact, this information is generally presented by the class lattice diagram of the class submodel. Nevertheless, inheritance diagrams describe more detailed information (features presented, access of features, inheritance links etc.) related to the focused classes.

This graphical approach for describing the class inheritance hierarchy structure was designed to maximize the amount of key information displayed in a minimum amount of space. Figure 4-11 shows a partial inheritance diagram for the example Employee Management described in Figure 4-2. The arrow from *Employee* to *Programmer* indicates that *Programmer* is a derived class of *Employee.* Inheritance is also transitive. If *Wage_Programmer* is a derived class of *Programmer* and *Programmer* is a derived class of *Employee,* then *Wage_Programmer* is a derived class of *Employee* (indirectly). *Employee* is therefore a direct base class of *Programmer* and *Programmer* is a direct derived class of *Employee.*

Inheritance diagrams identify the features of the base classes and derived classes that are displayed. They reveal; (1) details about the implementation (by feature symbols); (2) inheritance of the features; and (3) their visibility within the system as well. *Programmer,* which is a direct derived class of *Employee* and an indirect derived class of *Person,* inherits *Employee*'s features in conjunction with the features *name* , *address*, and *SSN* defined in *Person. Investor* has inherited features from both the class *Employer* and *Person.* Since none of these inherited features are redefined, they do not appear in the *Investor* class box.

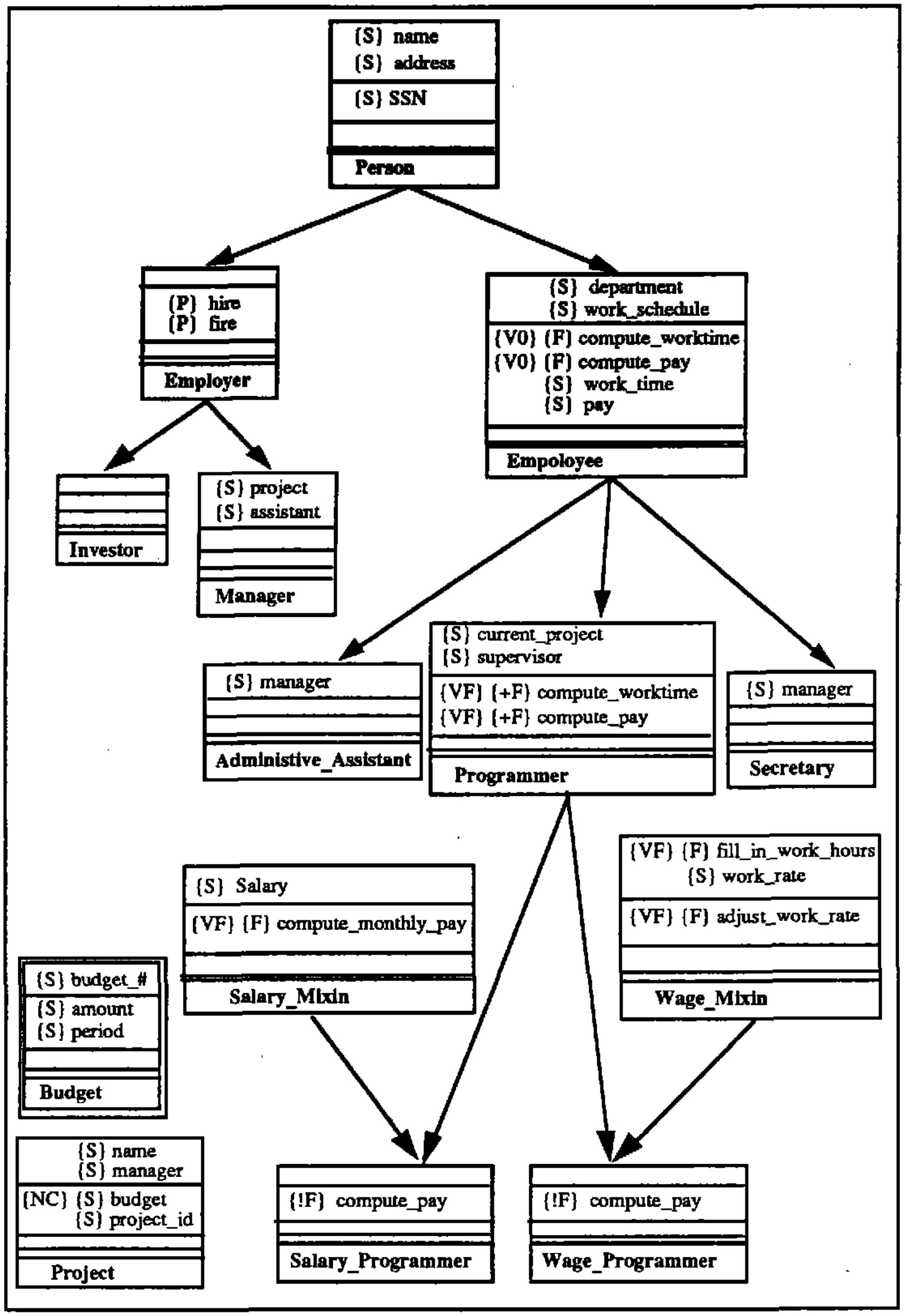


Figure 4-11. A Partial Inheritance Diagram of Employee Management System.

The symbol 'V0' is added to functions *compute_worktime* and *compute_pay* in class *Employee*, indicating that these functions are to be implemented as C++ pure virtual functions. Both features are further entitled as virtual functions with the symbol 'VF' in the class *Programmer*. *Compute_pay*, which is refined as general member functions ('F') in both *Salary_Programmer* and *Wage_Programmer*, reveals where the actual implementation takes place. In *Programmer*, the plus sign (+) preceding *compute_worktime* and *compute_pay* indicates that these features have additional constraints that specify the contracts inherited from the base class. The symbol '!' prefixing *compute_pay* presented in *Salary_Programmer* and *Wage_programmer* indicates that the constraints hold shadow the contracts inherited from *Programmer*.

Wage_Mixin is implemented as an abstract base class for any wage-paid employees. The member functions in this class, such as *adjust_work_rate* and *fill_in_work_hours*, are declared as virtual functions for possible further refinement in its derived classes (More details of abstract base class are discussed in Section 6.4).

The class box for describing a nested class is different from general classes in that it uses a double class box, as the class *Budget* illustrated in the figure.

4.3.6 Class Invariant Data Sheets

In IDEF4/C++, each class has an associated specification for its class definition. This specification is documented in the associated class invariant data sheet (CIDS). CIDSs describe; (1) the definitions of features and behaviors that individual instances of the class must possess; (2) the types of the direct inheritance links; and (3) the class invariant constraints which must always be maintained as true.

One purpose of CIDSs is to provide documentation for those who will maintain the installed system and for those who will implement the design. As shown in Figure 4-1, IDEF4/C++ diagrams are centered on the classes defined for the system. A CIDS includes numerical identifiers for referencing the diagrams associated with the named class. The names of directly present features of a class are also included to allow reference to the other design components such as method taxonomy diagrams via dispatch mapping.

Generally, computation-initiating features are grouped and presented in CIDSs according to their behaviors. The description of each feature, including the name of the feature, the kind of the feature (generic feature, attribute, routine, slot, function, and procedure), and the feature access (public, protected, and private), is also captured in CIDSs. Virtual features can be accessed via the listing of the inheritance links to the direct base classes. Constructors and destructors are also documented in CIDSs. In addition, check boxes are used to indicate whether the class is to be implemented as a struct, a union, a class template, or a class.

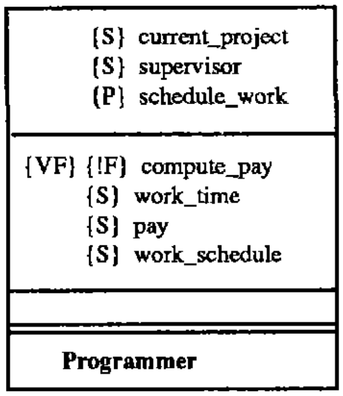
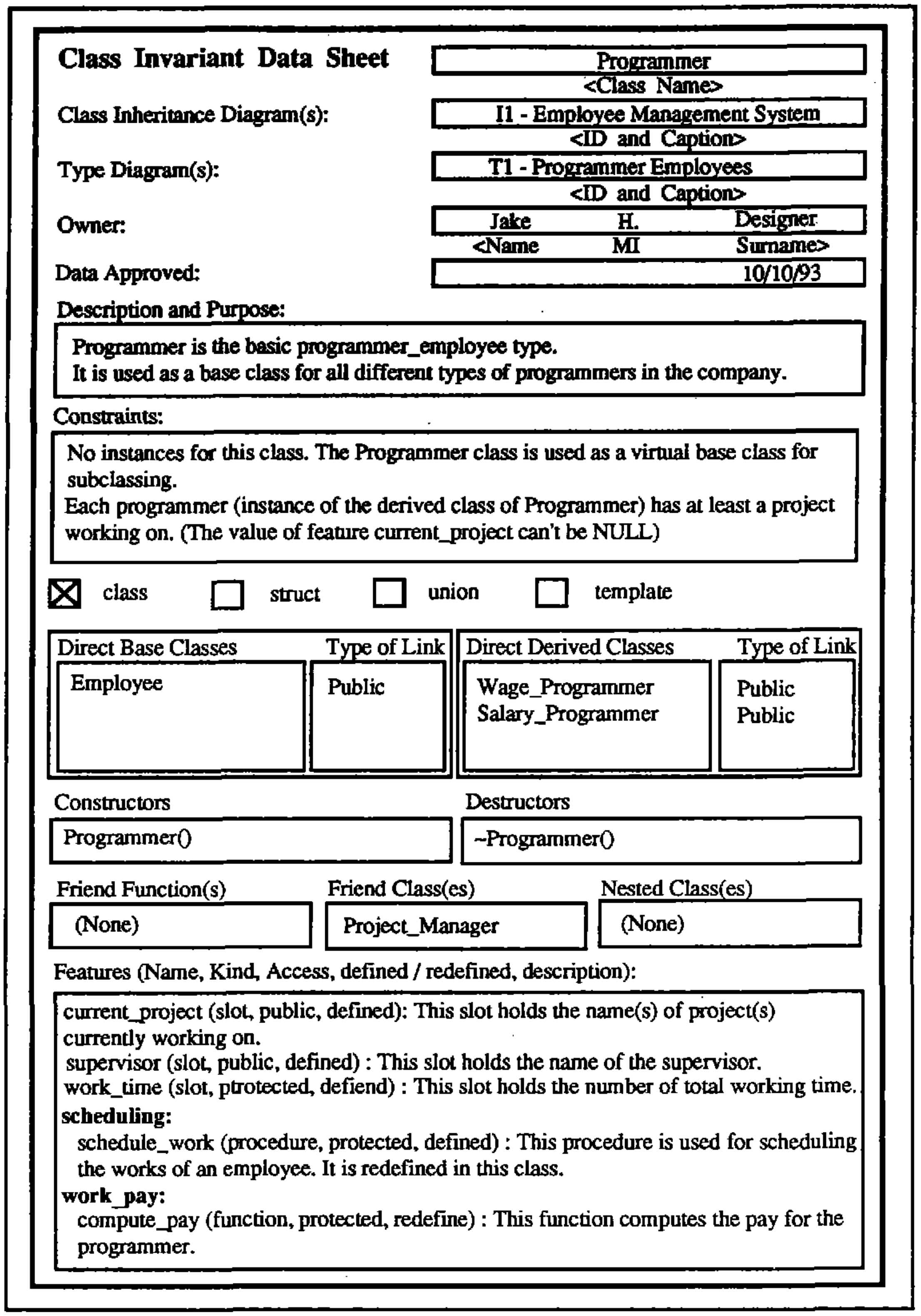


Figure 4-12. The Programmer Class.

Figure 4-12 describes a *Programmer* class. Figure 4-13 presents an example CIDS for the class *Programmer*. The name of the class and other relevant bookkeeping information is identified at the top of the data sheet. IDs and captions of the associated inheritance diagrams and type diagrams are specified for reference purpose. Constraints on the implementation of the class can be described in plain English (as shown in the example), first order logic, or other languages suitable for expression. By providing the list of direct base classes, the implementor can locate those inherited features. If the class *Programmer* were to be deleted or modified, the direct base and derived class lists would provide those modifications to the design with the means to quickly trace which classes in the system would be affected by the change.

The list of directly present features is also presented in the CIDS. For each directly present feature of *Programmer*, the CIDS will contain the name, feature type, feature access etc., that can be found in the *Programmer* inheritance diagram. Behavior specifying features (routines/functions/procedures) are grouped according to their behavior types. For example, *compute_pay* is grouped under the *work_pay* behavior. CIDSs are the only place in the design in which the textual definition of features is provided.



Class Invariant Data Sheet

	Programmer
	<Class Name>
Class Inheritance Diagram(s):	I1 - Employee Management System
	<ID and Caption>
Type Diagram(s):	T1 - Programmer Employees
	<ID and Caption>
Owner:	Jake H. Designer
	<Name MI Surname>
Data Approved:	10/10/93

Description and Purpose:

Programmer is the basic programmer_employee type.
It is used as a base class for all different types of programmers in the company.

Constraints:

No instances for this class. The Programmer class is used as a virtual base class for subclassing.
Each programmer (instance of the derived class of Programmer) has at least a project working on. (The value of feature current_project can't be NULL)

☒ class ☐ struct ☐ union ☐ template

Direct Base Classes	Type of Link	Direct Derived Classes	Type of Link
Employee	Public	Wage_Programmer	Public
		Salary_Programmer	Public

Constructors	Destructors
Programmer()	~Programmer()

Friend Function(s)	Friend Class(es)	Nested Class(es)
(None)	Project_Manager	(None)

Features (Name, Kind, Access, defined / redefined, description):

current_project (slot, public, defined): This slot holds the name(s) of project(s) currently working on.
supervisor (slot, public, defined) : This slot holds the name of the supervisor.
work_time (slot, ptrotected, defiend) : This slot holds the number of total working time.
scheduling:
schedule_work (procedure, protected, defined) : This procedure is used for scheduling the works of an employee. It is redefined in this class.
work_pay:
compute_pay (function, protected, redefine) : This function computes the pay for the programmer.

Figure 4-13. Class-invariant Data Sheet for Class Programmer.

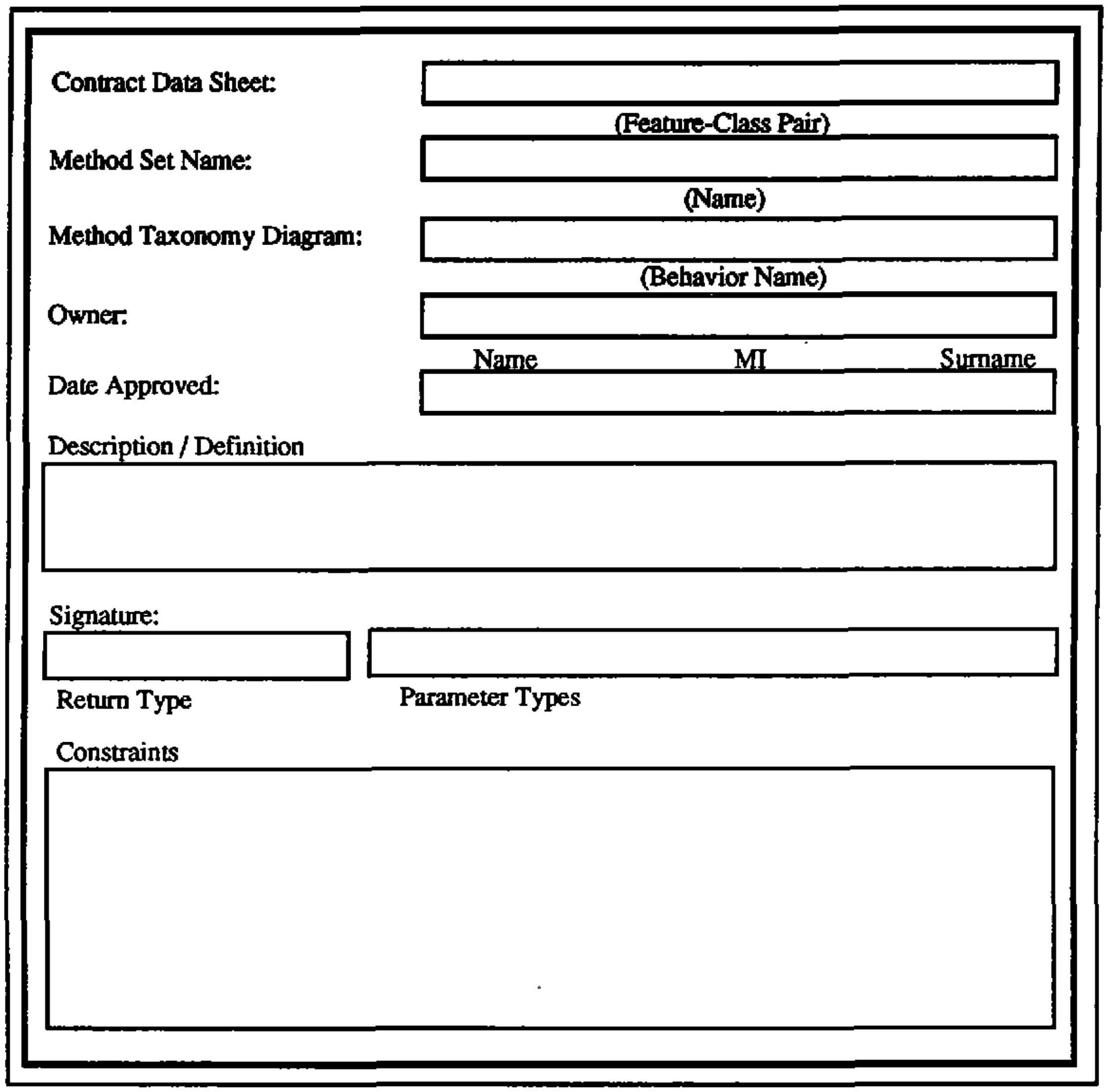
4.4 Method Taxonomy Diagrams

As inheritance diagrams arrange classes in a generalization (specialization) hierarchy, method taxonomy diagrams arrange method sets in the same way according to their contracts.

In IDEF4/C++, a method is any implementation (i.e., a C++ member function) that satisfies the contract for the method set. A method set is completely determined by its contract and logically equivalent contracts pick out identical method sets. Just as the class-invariant constraints hold for all instances of a class, the contract for a method set is invariant for all the methods in the set.

Contract of a method set is documented in the associated Contract Data Sheet (CDS) as shown in Figure 4-14. One aim of using CDs is to facilitate communication and coordination between designers and programmers in a large software project since documenting method contracts confirms the same expected system behavior as designed and implemented.

With a CDS, one can easily locate associated diagrams by using the feature name and its defining class name. The class name refers to the CIDS of the class and the CIDS can be used to locate the associated type diagram and inheritance diagram. The generic behavior that groups the feature refers to the method taxonomy diagram that classifies all the similarly-behaved features in the system, such as the *work_pay* behavior documented in the class *Programmer* 's CIDS as shown in Figure 4-13. However, for quick reference, the name of this method taxonomy diagram is also documented in the CDS (see Figure 4-14). The combination of the feature and class name allows one to reference the client diagram that applies to the method set. Moreover, the signature of the feature is also documented in the CDS, which includes the return type and the types of the parameters of the feature. This information is critical for implementing the method set.



Contract Data Sheet:

(Feature-Class Pair)

Method Set Name:

(Name)

Method Taxonomy Diagram:

(Behavior Name)

Owner:

Name MI Surname

Date Approved:

Description / Definition

Signature:

Return Type

Parameter Types

Constraints

Figure 4-14. Contract Data Sheet.

As mentioned previously, method sets in a design are grouped together by related contracts (similar behaviors) to form a method taxonomy for a particular type of system behavior. In other words, each method taxonomy diagram identifies a generic system behavior. Therefore, by convention, the name of a method taxonomy diagram is the generic behavior that is being described. Following the previous Employee example, we describe the *work_pay* generic behavior in terms of a method taxonomy diagram illustrated in Figure 4-15. The boxes represent method sets and arrows specify additional constraints (+) or redefinitions (!). The arrows point from the less-specific to more-specific method sets. A method taxonomy diagram may be arranged either from left to right or top to bottom for the most-general to the most-specific method set. In the diagram, the constraints on the method set *workpay* indicate that the methods in the set will calculate the workpay of any person who is classified as an employee. The other method sets in the diagram represent specializations of the constraints placed on the first method set. *Pay-by-hour-default-rate* and *pay-by-hour-special-rate* will calculate the workpay only for the employees who are wage-paid. Method set *pay-by-month* specifies the methods that will calculate the workpay only for the employees who are monthly-paid. In both cases, the new or additional constraint supersedes or specializes the contract on the method set *workpay*, requiring a more restrictive or specialized type of behavior.

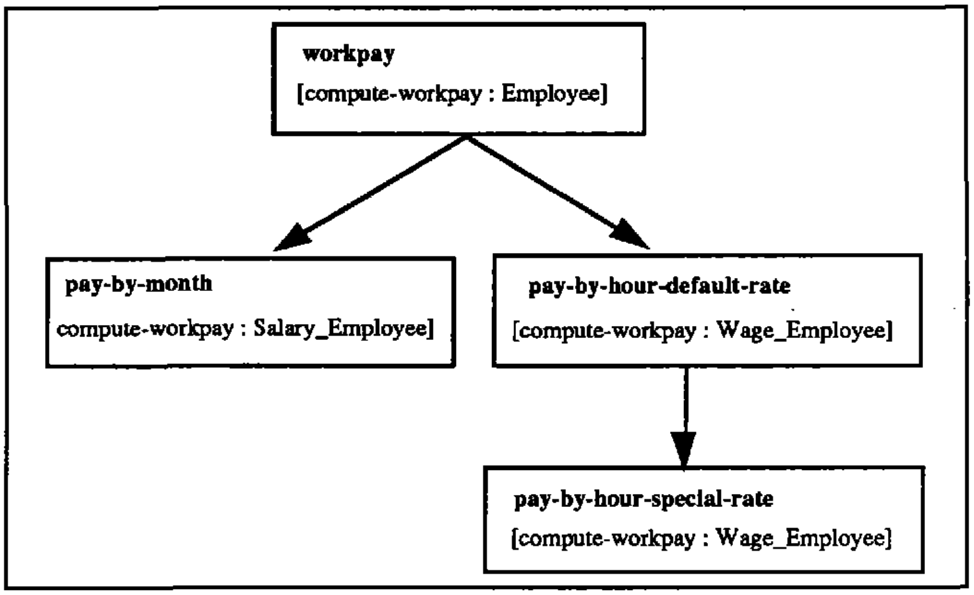

workpay
[compute-workpay : Employee]
pay-by-month
compute-workpay : Salary_Employee]
pay-by-hour-default-rate
[compute-workpay : Wage_Employee]
pay-by-hour-special-rate
[compute-workpay : Wage_Employee]

Figure 4-15. Work_pay Method Taxonomy Diagram with Dispatching Mapping.

Figures 4-15 and 4-16 together illustrate how method taxonomy diagrams are referred to from other components of the design using explicit dispatch mappings. Dispatch mappings must be explicitly defined when more than one feature with same name is defined in a class.

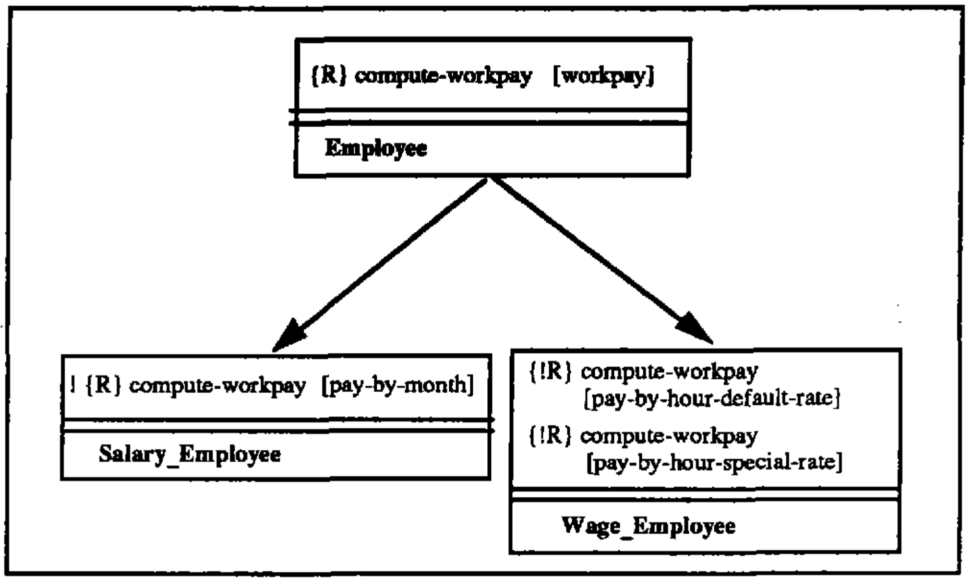

{R} compute-workpay [workpay]
Employee
! {R} compute-workpay [pay-by-month]
Salary_Employee
{!R} compute-workpay [pay-by-hour-default-rate]
{!R} compute-workpay [pay-by-hour-special-rate]
Wage_Employee

Figure 4-16. Employee Inheritance Diagram with Dispatch Mapping.

Figure 4-16 illustrates that in the class *Salary_Employee*, the routine *compute-workpay* is redefined and will be dispatched to the method set *pay-by-month* described in Figure 4-15. This dispatch matching is specified by using '[ ]' in both diagrams. The term "dispatched" refers to the way of indicating which method-set contract is associated with the computation-initiating feature and its class. Two *compute-workpay* routines are redefined in *Wage_Employee*: one is dispatched to the method set *pay-by-hour-default-rate* , the other is dispatched to *pay-by-hour-special-rate*. In fact, these two *compute-workpay* routines specify the use of overloaded functions in C++ as both are dispatched to different method sets in the same method taxonomy diagram. This example shows that the method taxonomy diagram does not necessarily group method sets in the same hierarchy that an inheritance diagram presents behavioral features. Without being dispatched to their related contracts, one can hardly specify the intent of overloaded functions.

Method taxonomy diagrams are important for designers as a means of classifying and organizing method sets in that they specify the common behaviors across a wide variety of systems as well as providing a catalog of previously coded methods for reuse. If a particular contract is very widely used and studied (e.g., sorting), the corresponding method set and its subsets may form quite a complex taxonomy. Such a taxonomy may serve as a reusable resource for designers.

4.5 Type Diagrams

Inheritance is generally considered as the primary relation between classes in object-oriented modeling. Yet many other interesting relations are established implicitly through the values of the attribute features in the classes as well. These relations are generally structured around the roles of particular objects in relation to other classes. These roles in IDEF4/C++ are specified by the attributes in the classes. Therefore, the related object type is often a key constraint for capturing the particular role that an object play in a particular relation. In IDEF4/C++, the management of these role-definition processes is accomplished through type diagrams. In the IDEF4/C++ discipline, each attribute has a return type which defines the type of return value. Type diagrams, as a part of class submodel, provide graphical and textual notations for displaying the return types of attributes of the classes.

Type diagrams are syntactically composed of class boxes and type links. Only features that have return values, such as attributes, functions, or slots, will be shown in the diagrams. There are four kinds of type links:

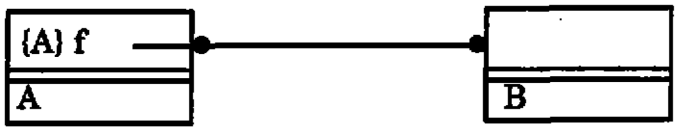


Figure 4-17. Single-valued Type Link.

- Single-valued type link

  Figure 4-17 illustrates the notation for a single-valued type link. A single-valued type link describes that the return value for an attribute is an instance of a class. In the figure, the return value of the attribute *f* is an instance of class *B*. The type of *f* is *B*.

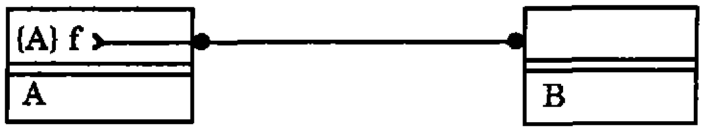


Figure 4-18. Multi-valued Type Link.

- Multi-valued type link

  A multi-valued type link describes that the return value for an attribute is composed of a structured collection of instances of a class. As illustrated in Figure 4-18, the values in the attribute *f* are a collection (i.e., a list) of instances of type *B*.

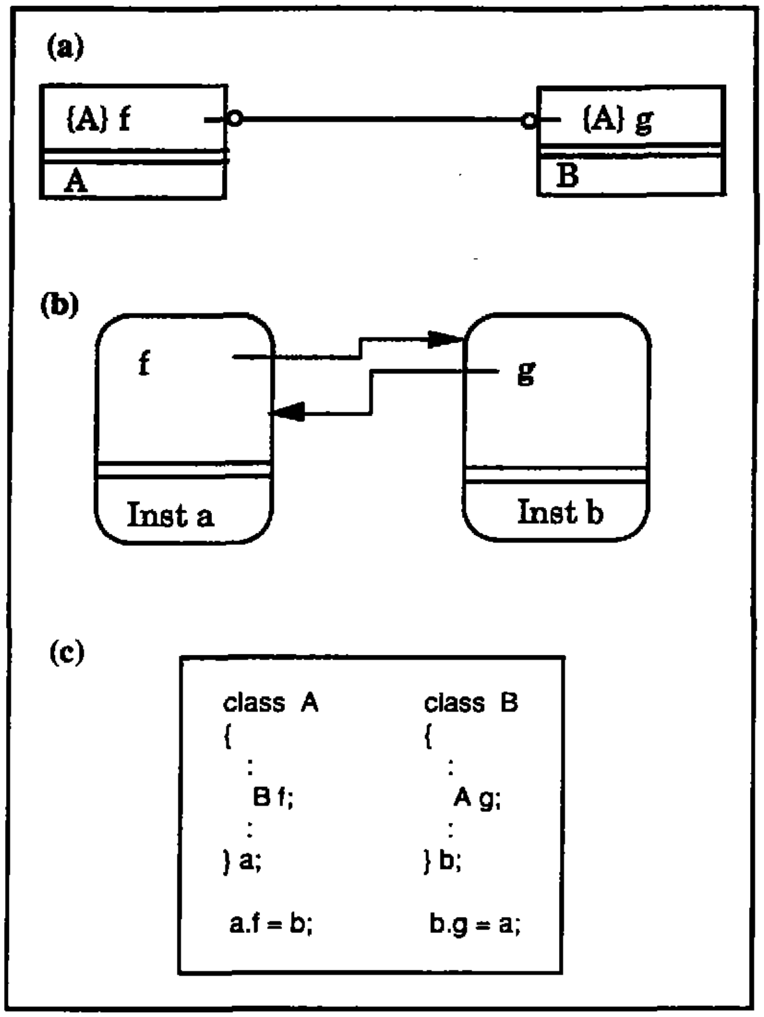
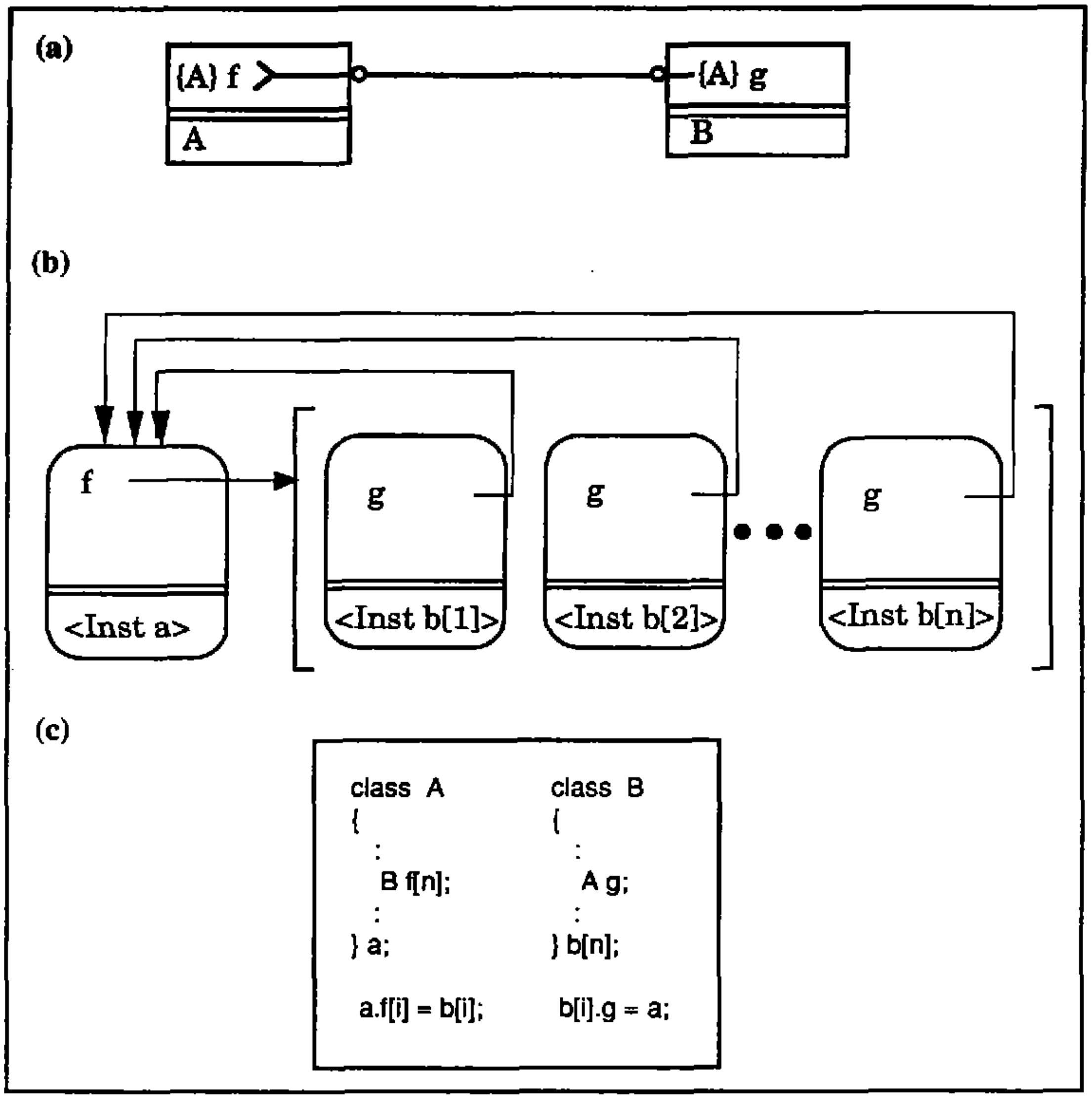


Figure 4-19. Single-valued Inverse Type Link.

- Single-valued inverse type link

Consider the example described in Figure 4-17. If an instance *b* of class *B* has a feature *g* and the return value of *g* is an instance of class *A*; and it is, in fact, just that instance which has *b* as the return value of its feature *f*, then we say the type link between *A* and *B* is single-valued inverse. Figure 4-19(a) illustrates the notation of this type of link. Figure 4-19(b) shows an instantiation diagram of

the example described above. We can think of the instances of class *B* as having "where used" pointers to the instances of class *A*. The C++ code example for these diagrams is illustrated in Figure 4-19(c).



```
class  A          class  B
{                 {
  :                 :
  B f[n];           A g;
  :                 :
} a;              } b[n];

 a.f[i] = b[i];    b[i].g = a;
```

Figure 4-20. Partial Inverse Type Link.

- Partial inverse type link

Partial inverses are the inverse relations other than one-to-one between classes. Figure 4-20(a) illustrates the notation

for a partial inverse type link. Instance *a* of class *A* has a feature *f* which holds a collection of instances of class *B*, and each instances in this collection has a feature *g* which holds exactly that instance *a* as the return value. This is also described in Figure 4-20(b), which shows that each instance of *B* points back to *a*. Figure 4-20(c) presents the C++ code example.

In a type diagram, feature return types can also be specified by concatenating the attribute's name with the return type. This alternative syntax is used in larger diagrams to reduce the unnecessary clutter of the diagram by eliminating a number of links. It can also be used by the designer to de-emphasize certain relations and focus the attention of the design reviewers on specific relations (i.e., those shown with links). This approach is typically used for common data types such as integer and Boolean. In IDEF4/C++, these common or user predefined types can be defined and collected in the Predefined Data Type List, as well as classes in third-party class libraries which are included in the design. For example, consider a user predefined type such as *String*, which is defined as the *char pointer* in C++:

```
typedef String char*;
```

The predefined type *String* will be collected in the Predefined Data Type List with its *typedef* definition, allowing for the use of type diagrams. These predefined types are placed behind a colon ':' after the named feature as shown in Figure 4-21(a). Another example shown in Figure 4-21(b) illustrates the use of the predefined type from third-party class libraries. Consider a design of the *Employee-data-entry-form* class as a part of the interface for the Employee Management

System. The class consists of two features with predefined types from the Borland ObjectWindows class library: *RTMessage* and *TCheckBox*. The predefined type *RTMessage* is defined as a pointer to the type *TMessage* which is a Borland message data structure. *TCheckBox* is a predefined class used for displaying and managing a check box as an input item of the data entry form. Again, both of them must be specified in the Predefined Data Type List prior to use, as illustrated in the example. Moreover, the CIDS for the *TCheckBox* and the CDSs for its member functions have to be documented properly in the design, although these CDSs would likely have only their signatures specified.

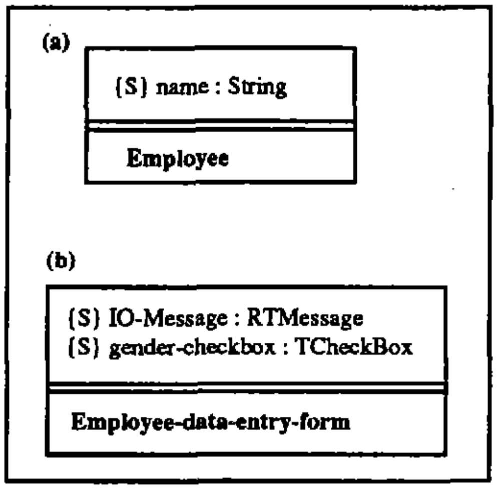


Figure 4-21. Textual Notation for Feature Return Types.

An example type diagram for the Employee Management System is illustrated in Figure 4-22. In the diagram, type links specify the intended relations between classes in the system, such that a vice president has an executive assistant, or both the *compute-workpay* functions of *Project-Manager* and *Programmer* will return a type of *Workpay*. A multi-valued link placed between class *Programmer* and class *Project* indicates that the slot *current-project* of *Programmer* will return a collection of instances of *Project*. This is done in a similar fashion as the feature

team-members **defined in the class** ***Project-Manager*****, whose return type is specified by a link pointing to class** ***Programmer*****.**

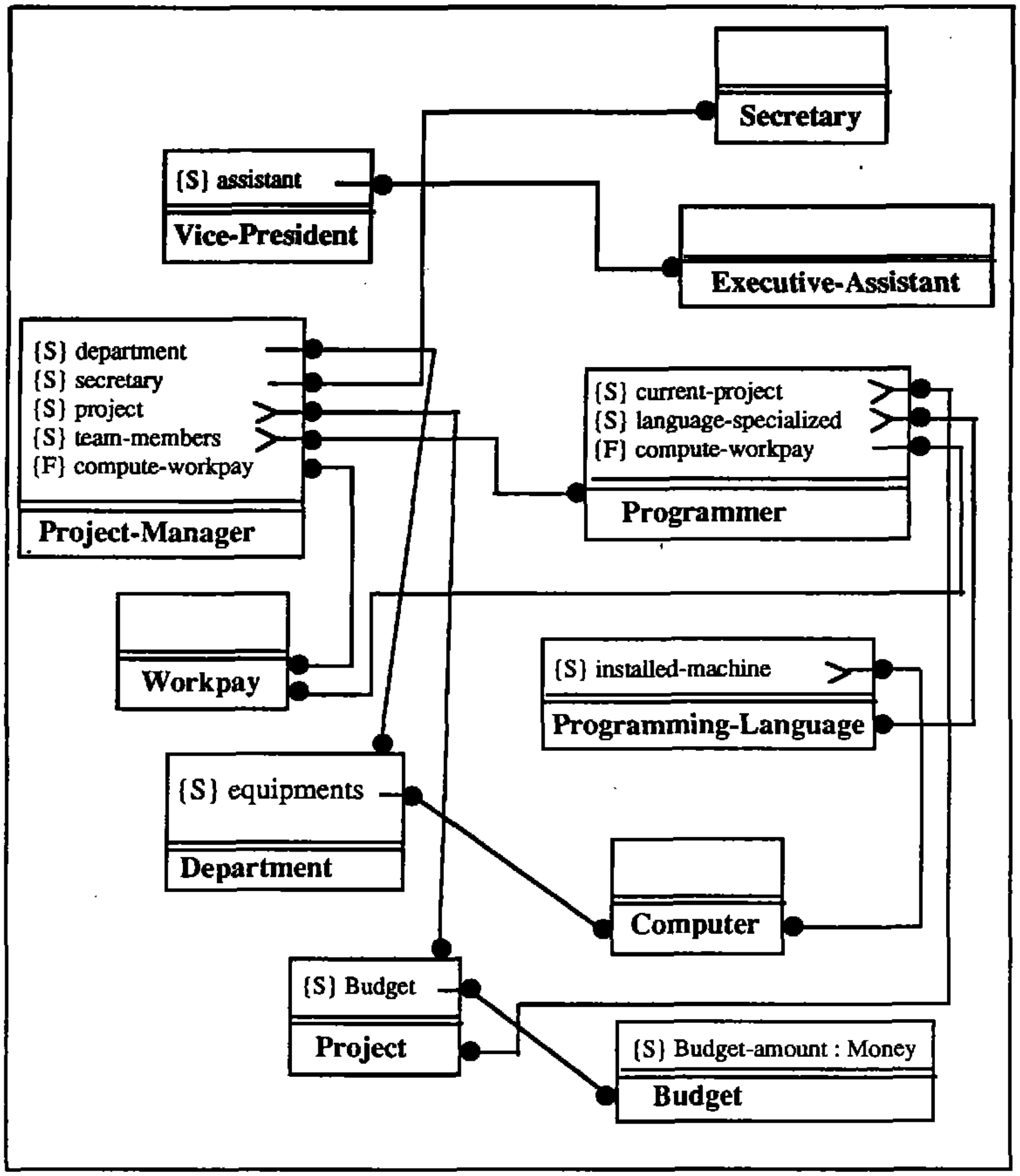


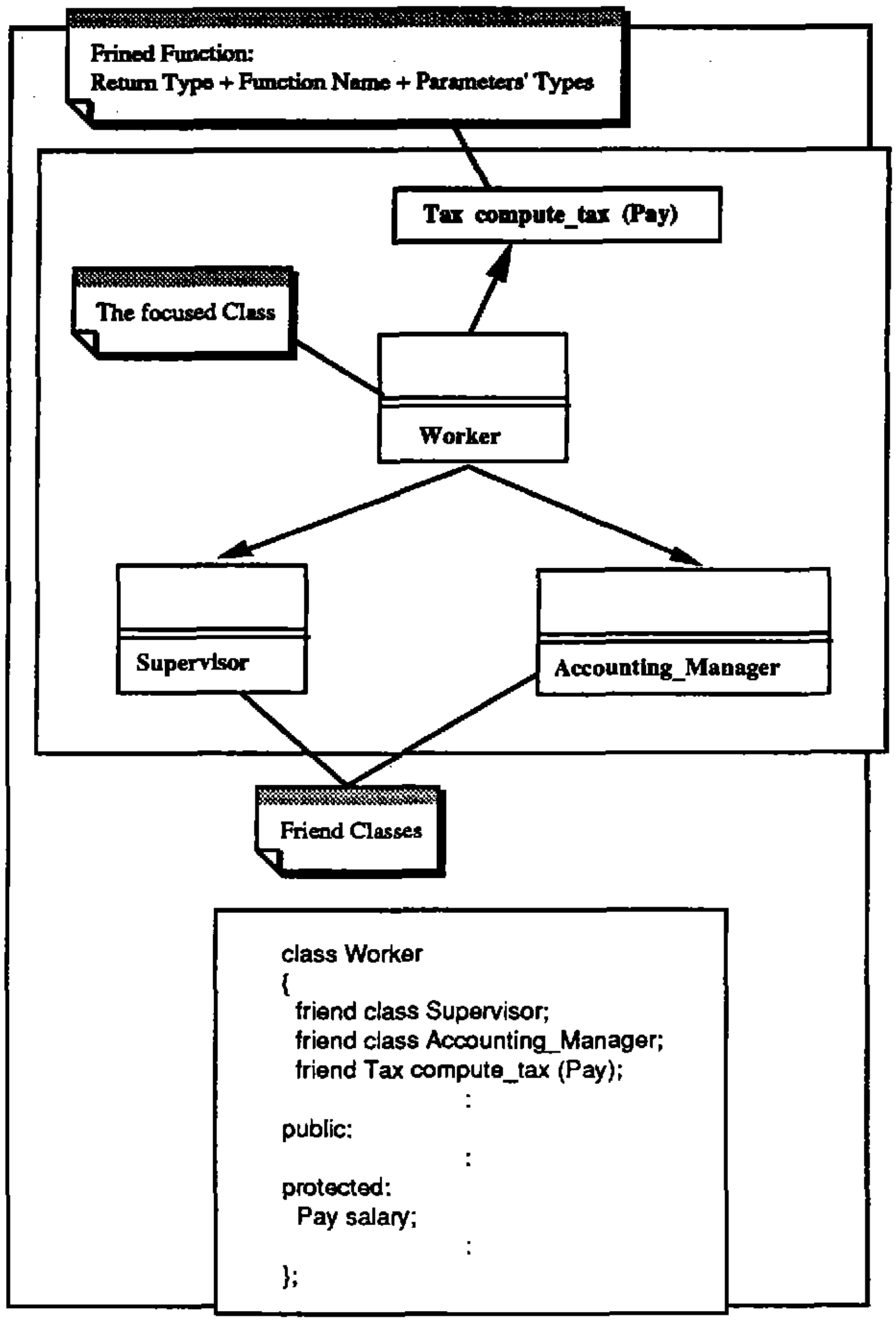
Figure 4-22. An Example Type Diagram for the Employee Management System.

4.6 Friend Diagrams

Friend diagrams, as extension to IDEF4, are used to show the C++ friend declaration relationships between the focused classes and their friend classes / functions.

Each friend diagram focuses on one class at a time, describing all the friend declarations within its definition. A friend diagram employs three diagram symbols: class boxes, friend function boxes, and arrows. The class boxes are the general class boxes as used in the IDEF4/C++ notations, which present both the focused class and its friend classes. However, only the class name will be shown on the class box; features and other unrelated class details won't be given in friend diagrams. Friend functions are presented in terms of the whole function definition by using the friend function boxes. A function definition includes: (1) the return type of the function (if none, use *void*); (2) the function's name; and (3) the list of the types of parameters. The arrows specify the friend declaration links, pointing from the focused class to its friend classes and functions. Friend functions are listed at the top of a diagram, where the friend classes are listed on the bottom.

Figure 4-23 gives an example of a friend diagram along with the C++ code, which describes the friend declarations defined in the definition of class *Worker*. The presentation is very straight-forward. In the figure, the friend class *Supervisor* and *Accounting_Manager* are displayed on the bottom of the diagram, where the friend function *compute_tax* is placed at the top. Friend declaration, as specialized in C++, is used to break the default encapsulation mechanism supported by the language. For example, both the friend function *compute_tax* and the instances of *Supervisor* and *Accounting_Manager*, are able to access to the protected or private features of *Worker*, such as *salary* through the friend declarations However, the use of friend diagrams really depends on the design intent with respect to other class relations in the system. One should be cautious in applying such declarations.



```
class Worker
{
  friend class Supervisor;
  friend class Accounting_Manager;
  friend Tax compute_tax (Pay);
                    :
public:
                    :
protected:
  Pay salary;
                    :
};
```

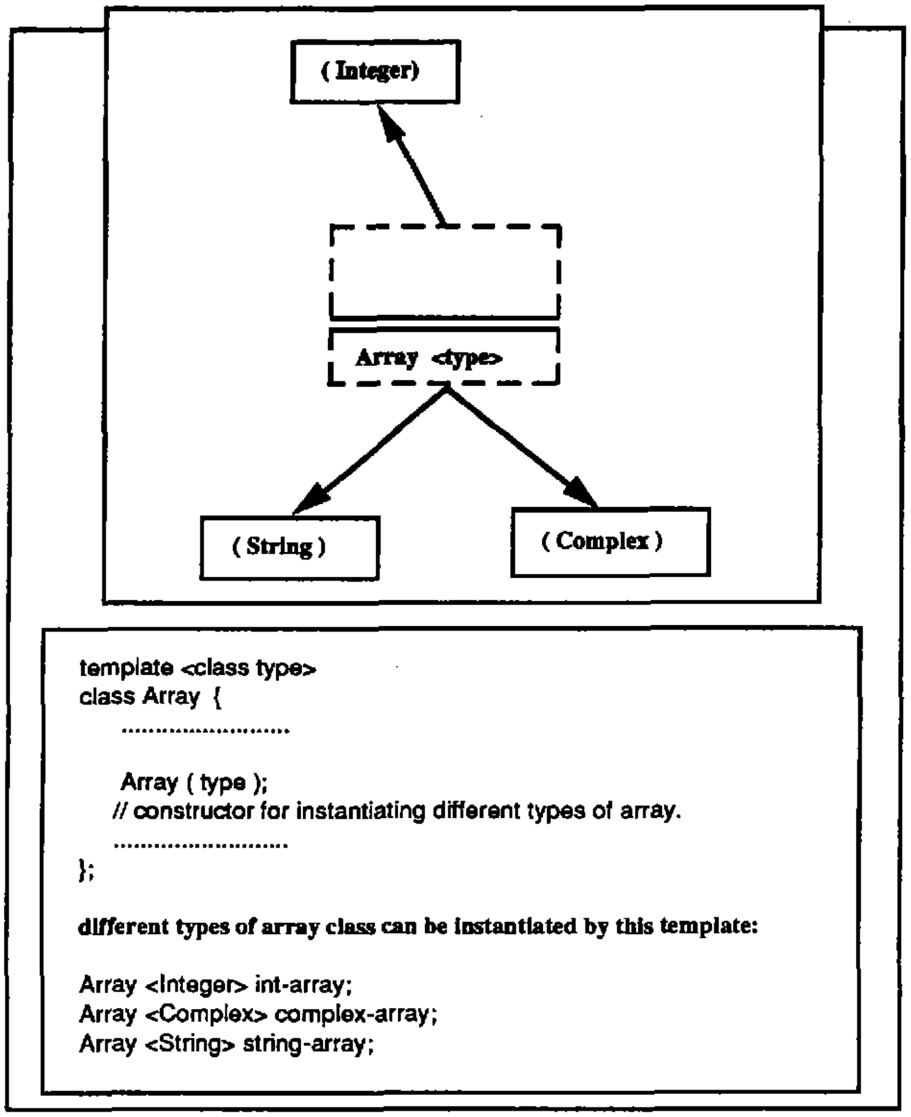
Figure 4-23. A Friend Diagram and C++ Code Example for Class Worker.

4.7 Template Diagrams

Template diagrams are used to specify the class template declaration in C++. A C++ template class is a parameterized class and consequently different types of parameters will allow different classes to be instantiated.

Template diagrams employ three diagram symbols: template class boxes, parameter type list boxes, and arrows. A template diagram focuses on one class template at a time, which is displayed in the center of the diagram. Template class boxes, as illustrated in terms of dotted class boxes, are used to present the focused class templates. Similar to the general IDEF4/C++ class box, features defined in the template would be shown in public, protected, and private groups. The template name and its parameter list are placed where the class name would usually be shown in the class box. The parameter type lists are presented by using single boxes. Arrows pointing from the template to the parameter type list boxes indicate the connection between the parameterized classes and their template.

For example, consider a C++ class template *Array* to be used as a template for different types of arrays, such as integer, string, and complex. The template diagram is illustrated in Figure 4-24 with the C++ code presented. In the diagram, the template *Array* has only one parameter, *type*, which is used to indicate the type of array to be instantiated. To instantiate an array class (*int-array*), one needs to "parameterize" the parameter *type* with the intended type provided (*Integer*). There are three different types of arrays illustrated in the diagram: *Integer*, *String*, and *Complex*. These types should be documented in the Predefined Data Type List and the detailed definition of the template *Array* should be documented in its CIDS.



```
template <class type>
class Array {
     .........................

      Array ( type );
     // constructor for instantiating different types of array.
     .........................
};
```

different types of array class can be instantiated by this template:

```
Array <Integer> int-array;
Array <Complex> complex-array;
Array <String> string-array;
```

Figure 4-24.
A Template Diagram and C++ Code Example for Class Template Array.

4.8 Client Diagrams

Client diagrams, as part of the method submodel, are used for algorithmic decomposition. They are the only IDEF4/C++ diagrams that specify, however abstractly, the internal structure of routines (computation-initiating features).

Figure 4-25 shows a sample client diagram for the routine *show-project-information* as defined in the class *Project-Manager.* Routines are shown along with their directly-defining class names, such as *Project-Manager:show-project-information.*. The links between boxes represent control references or "subroutine calls" from one routine (as the *client*) to another (as the *supplier*). For example, the link between *show-project-information* and *print-project* specifies that the implementation for *show-project-information* (the client) calls the feature *print-project* defined in the class *Project* (the supplier).

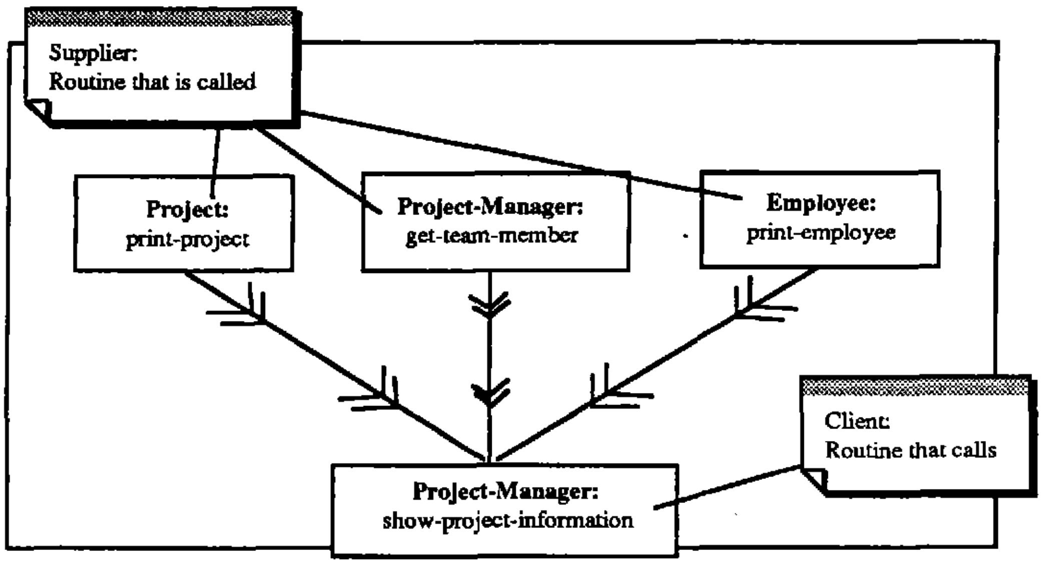


Figure 4-25.
Client Diagram for Show-project-information of Class Project-Manager.

Figure 4-26(a) displays the C++ code for *Project*, *Project-Manager*, and *Employee*. Each class defines the member functions to be called by *show-project-information*, as indicated in Figure 4-25. Figure 4-26(b) presents the code for these calling algorithm. To show project information, the client calls *print-project* defined in *Project* first and then calls the *get-team-member* defined in *Project-Manager* to get the members in the project team. Next, for each member (which is an instance of *Employee*), the function *print-employee* is called. In fact, the diagram specifies that the implementation for the client (*show-project-information*) will call each supplier directly, not some generic functions. If the class associated with the client function has not been specified in the diagram[11], dispatching will occur at run time for any implementation. For an implementation in C++, this would indicate the need for a dynamic binding for *print-project*. This design issue is discussed in Section 6.5.

```
(a)   class Project {                    class Employee {
      friend  Project-Manager;           friend Project-Manager;
      public:                            public:
        ..........................         .......................
      protected:                         protected:
        Budget  prj-budget;                Pay  salary;
        void  print-project(void);         void print-employee (void);
        ..........................         .......................
      };                                 };

      class Project-Manager {
      public:
        Project *get-project(void);
        ..........................
      protected:
        void  show-project-information (Project *);
        Employee *get-team-member (Project *);
        ..........................
      };

(b)   void  Project-Manager::show-project-information (Project *prj)
      {
        Employee *person;

        prj->print-project();
        while (person = get-team-member(prj);)
        {
          person->print-employee();
          .....................
        };
          .....................
      }
```

Figure 4-26. C++ Code Example for Show-project-information.

[11] For instance, no class name shown before *print-project*.

4.9 Instantiation Diagrams

The purpose of instantiation diagrams is to facilitate the development of test case scenarios. Test case scenarios, in turn, are used to validate the design and document examples of the intended design. Ultimately, this validation process aids programmers in implementing the design. An instantiation diagram looks much like a type diagram. It uses a round-cornered box to represent instances of a class, analogous to the IDEF4/C++ class box. For example, the instantiation of an employee named *John* from the class *Employee* (see the following code) has "*<Employee John>*" as its unique identifier. The representation of instance *John* is illustrated in Figure 4-27.

```
        Employee  John;
   ,or
        Employee *John = new Employee();
```

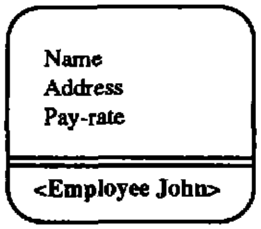
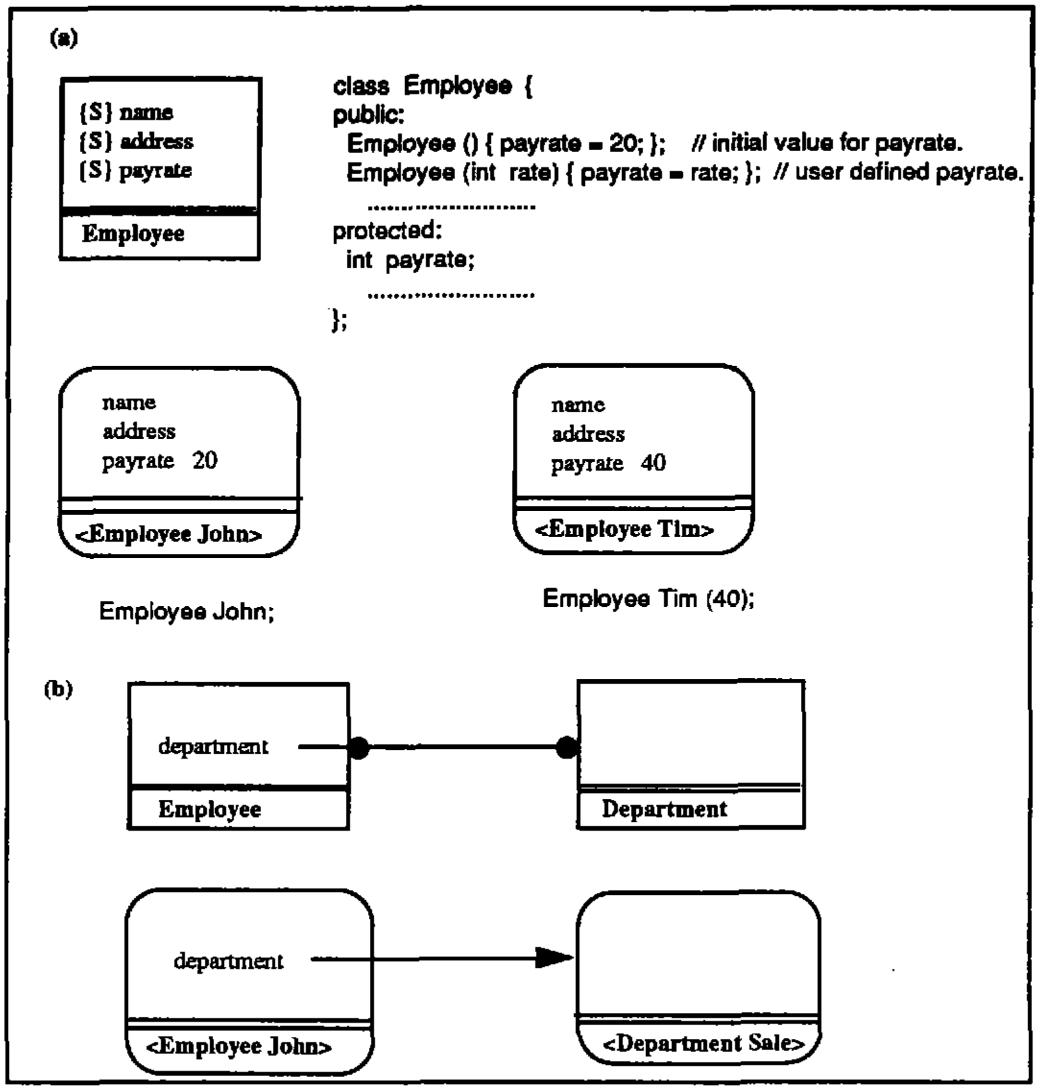


Figure 4-27. An Instance Box - John.

The instance attributes are listed in the upper region of the instance box. Instantiation diagrams provide two ways to indicate the value assigned to or returned by the attributes: directly displaying the value to the right of the attribute (Figure 4-28(a)) or using value links (Figure 4-28(b)). In the case of attributes of numeric type, it is acceptable to directly present the value in the box. The value links used in instantiation diagrams are presented in terms of arrows, pointing from an attribute to the class that is used as the return type. They start inside the instance box, next to the attribute whose value is being annotated, and end in an arrow pointing to the boundary of an instance box as shown in the figure.



```
class Employee {
public:
  Employee () { payrate = 20; };   // initial value for payrate.
  Employee (int rate) { payrate = rate; }; // user defined payrate.
   ........................
protected:
  int payrate;
   ........................
};
```

```
Employee John;
```

```
Employee Tim (40);
```

Figure 4-28. Example Instantiation Diagrams.

4.10 Dispatch Mapping

In an IDEF4/C++ design, it is possible for a generic behavior to have more than one computation-initiating feature specialized for that generic behavior in the same class (i.e., features of the same generic behavior group in a CIDS) or in different classes (i.e., features of the same generic behavior group but in different CIDSs). In other words, in a method taxonomy diagram, more than one method set will be associated with the generic behavior described by the diagram and different features in different classes may be implemented by them. Dispatch mapping, a connecting mechanism between the class submodel and the method submodel, is used to refer to the association between the computation-initiating **feature** and their related **method sets**.

In Section 4.4, we have described a *work_pay* method taxonomy diagram and an *Employee* inheritance diagram. We continue the discussion by using these diagrams to show the mapping between those behavioral features and method sets. Figure 4-29 depicts the dispatch mapping. Note that in the inheritance diagram, these *work_pay* behavioral features are all named as *compute-workpay*. However, contracts to be applied to their implementations are different. *Employee*'s *compute-workpay* is mapped to the *workpay* method, which is the most general method set in the method taxonomy diagram. *Salary_Employee*'s *compute-workpay* is mapped to *pay-by-month* and two *compute-workpays* in *Wage_Employee* are mapped to *pay-by-hour-default-rate* and *pay-by-hour-special-rate* separately.

In an IDEF4/C++ design, features with the same name but in different classes indicate a design for the C++ virtual functions. Features with the same name but defined in the same class indicate a design for the C++ function overloading. However, detailed definitions (i.e., parameter types and the number of parameters) of these overloaded functions can only be revealed from the association of their CDSs with the method sets that they are mapping to.

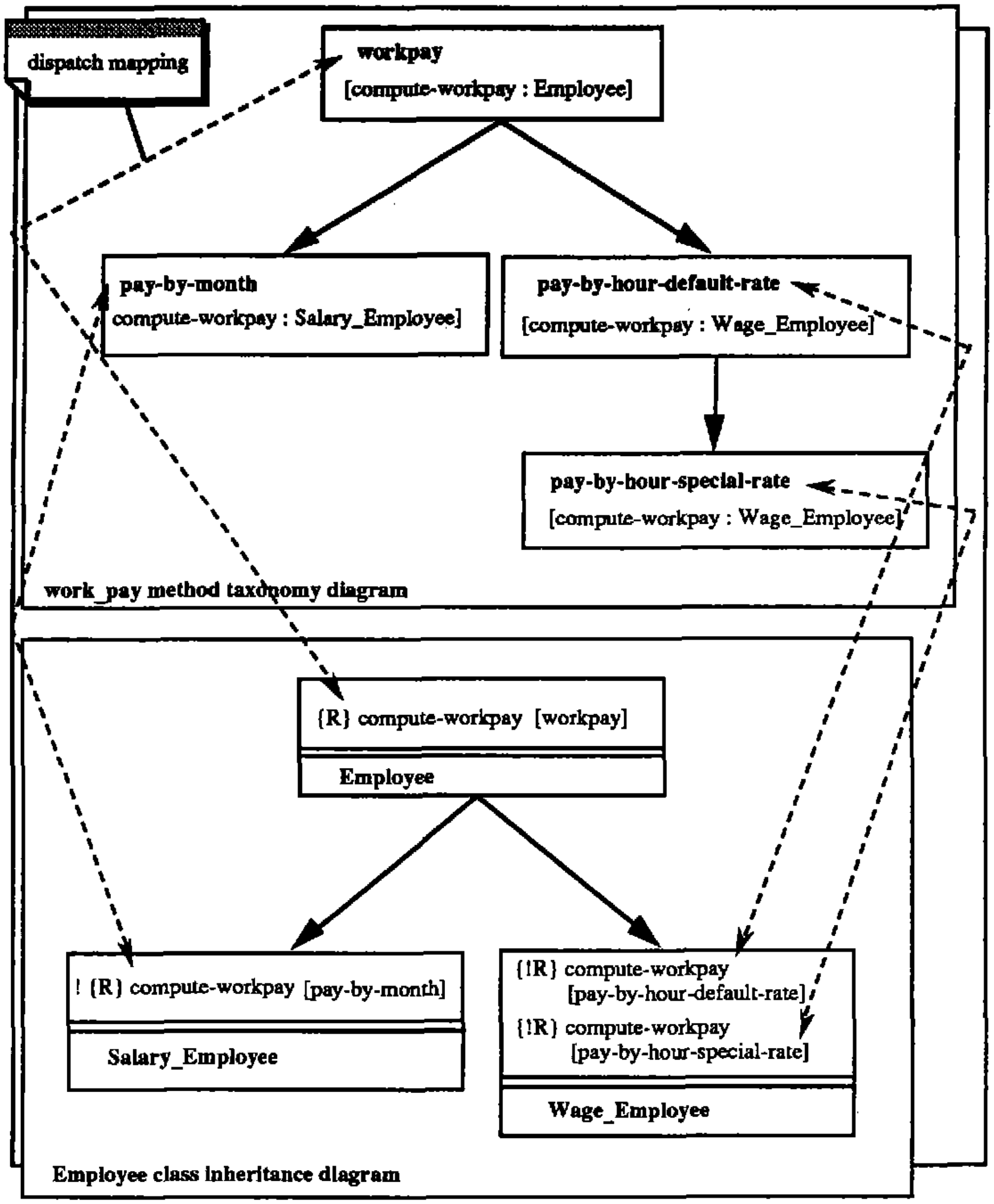

dispatch mapping
workpay
[compute-workpay : Employee]
pay-by-month
compute-workpay : Salary_Employee]
pay-by-hour-default-rate
[compute-workpay : Wage_Employee]
pay-by-hour-special-rate
[compute-workpay : Wage_Employee]
work_pay method taxonomy diagram
{R} compute-workpay [workpay]
Employee
! {R} compute-workpay [pay-by-month]
Salary_Employee
{!R} compute-workpay [pay-by-hour-default-rate]
{!R} compute-workpay [pay-by-hour-special-rate]
Wage_Employee
Employee class inheritance diagram

Figure 4-29. Dispatch Mapping of Work_pay Behavior.

CHAPTER V

METHOD PROCEDURE

5.1 Introduction

Perhaps one of the greatest challenges of developing an object-oriented design method is to define a development procedure. The purpose of such a procedure is the organization of the design artifacts and activities, especially for administrative purpose. Although the object-oriented design process is iterative, a development procedure defines a set of ordered activities allowing multiple developers or development teams to communicate across the design process and supports change control in the evolution of a design.

In the general object-oriented design process, there is a tension between the use of class decomposition (inheritance), object composition (aggregation), algorithmic decomposition, and polymorphic decomposition (message dispatching) (Mayer 92a). The "least commitment" philosophy employed by IDEF4 supports all four of these design perspectives, allowing one to refine a design seamlessly over the design evolution.

In this Chapter, we will address the issues of the transformation from generic IDEF4 to IDEF4/C++ and present a design procedure for IDEF4/C++. We will also provide an IDEF3 dynamic model of system development process which involves multiple developers / development teams, with the consideration of configuration management (change / version control).

5.2 Transformation from Generic IDEF4 to IDEF4/C++

IDEF4/C++, as a language dependent design method, extends several method features from IDEF4. However, the IDEF4/C++ implementation design method is not intended to replace IDEF4, a generic design method which creates language independent designs. By employing an IDEF4 generic design as the initial design for IDEF4/C++, the evolution of the design process therefore can be conducted seamlessly since both IDEF4 and IDEF4/C++ are derived from similar method concepts (as described in Chapter III). In general, the implementation design is a process of adding implementational details to the generic design model. The refinement is basically based on the language dependent features supported by the extended method. Developers therefore follow the design specification and create module prototypes by using the targeting language.

In this Section, we categorize the extended language dependent features provided by IDEF4/C++. These extended features, which were presented and discussed separately in the previous chapters, fabricate a set of transformation guidelines for the evolution from an IDEF4 generic design to an IDEF4/C++ implementation design. Table 5-1 in the following page summarizes these transformation features.

The table presents the basic transformation focus while evolving an IDEF4 design to an IDEF4/C++ design. Some features shown in the table such as class feature access, inheritance type link, and class feature symbols, have their own presentations in both methods. These transformation focuses have to be specified in more details in the evolution from IDEF4 to IDEF4/C++. Some other focuses are new to IDEF4 such as constructor / destructor, friend / template declarations, and class variables / operations etc.. They are specified in the implementation design process when needed. Routine signatures, which are necessary for coding method sets, have to be defined in the CDSs of the IDEF4/C++ design model before any implementation.

Transformation Focus	**Presentation in IDEF4**	**Method Element Presented**	**Presentation in IDEF4/C++**	**Default Transformation**
Inheritance Link Type	N/A	CIDS	virtual, Public / Protected / Private	Public
Class Feature Access Control	Public / Private	Class Box, CIDS	Public / Protected / Private	Public(in IDEF4) -> Public (in IDEF4/C++), Private (in IDEF4) -> Protected (in IDEF4/C++)
Class Feature Symbols	'A', 'R', 'S', 'F', 'P'	Class Box	'A', 'R', 'S'(slot), 'F', 'P', 'VF', 'V0', 'C', 'S'(static), 'NC'	If a feature is specified by 'F' or 'P' in IDEF4, the 'VF' symbol will be added by default in IDEF4/C++.
Constructor / Destructor	N/A	Class Box, CIDS	(1) procedures showing in the class box with class name as the procedure name. (2) they are also specified in CIDS constructor / destructor lists.	N/A
Implementation of Class	N/A	CIDS	Class / Struct / Union / Template	Class

Table 5-1. Transformation from IDEF4 to IDEF4/C++.

Transformation Focus	**Presentation in IDEF4**	**Method Element Presented**	**Presentation in IDEF4/C++**	**Default Transformation**
Friend Function / Class	N/A	Class Lattice Diagram, CIDS, and Friend Diagram	(1) presented by '=>' in class lattice diagrams. (2) specified in the CIDS friend list. (3) A friend diagram has to be provided for the class which has friend functions / classes defined.	N/A
Nested Class	N/A	Class Lattice Diagram, CIDS	(1) presented by '--->' in class lattice diagrams. (2) specified in CIDS nested class list.	N/A
Class Variables / Operations	N/A	Class Box	Specified by 'S' (static) symbol	N/A
Parameterized Class	N/A	Template Diagram	Presented as double-lined class box	N/A
Routine Signature	N/A	CDS	Specified in IDEF4/C++ CDS	N/A

Table 5-1. (continued)

5.3 IDEF4/C++ Design Development Activities

Development of an IDEF4/C++ design involves the creation of diagrams. Diagrams provide different perspectives for describing artifacts (classes, features, and method sets etc.) and the relationships among them. In general, the evolution of an IDEF4/C++ design is an iterative process of partitioning, classifying / specifying, merging / eliminating, and rearranging these design artifacts. These operations are employed in most of the design activities and might involve the creation / modification of different diagrams in each. The following steps present the design activities that are performed throughout the IDEF4/C++ design procedure.

- Analyze evolving system requirements.

  System requirements may or may not evolve through the design process. However, examining the user evolving requirements over time keeps the design on "the right track". User evolving requirements result in the occurrence of new or additional constraints to the design. Such constraints might therefore promote the need for creating new design artifacts or modifying the working versions of the design artifacts. This will certainly cause another design iteration. Moreover, if functional (such as IDEFØ), informational (such as IDEF1, IDEF1X), or process (such as IDEF3) models are available, they can be used as inputs to this activity.

- Develop and refine class hierarchy.

The development / refinement of the class hierarchy involves (1) detailing / rearranging design artifacts such as classes and features, (2) specifying the relationships between classes, and (3) refining the class-invariant constraints. This activity includes the following steps:

- Develop, refine or update class lattice diagram(s).
- Develop, refine or update inheritance diagrams.
- Create, refine features defined in the updated classes.
- Refine or update CIDSs.
- Create, refine or update friend diagrams as needed.
- Create, refine or update template diagrams as needed.

- Develop and refine class composition structure.

Composition relationship (aggregation) between classes is specified in type diagrams. The type links connect the value-returning features and the classes which are specified as the return types. For those types predefined by users (most likely the primitive data types such as *int, double* etc.) or provided by class library venders, the user predefined type list is to be used and further refinement might be needed. Note that if the changes affect design such as class definition or class relationship, the activities described in prior steps are involved. This activity includes the following steps:

- Develop and refine type diagrams.
- Refine and update user predefined type list.
- Update CIDSs as needed.
- Update inheritance diagrams or lattice diagram(s) as needed.

- Develop and refine method taxonomy.

  This activity involves design artifacts such as method sets, contracts, and generic behaviors. Method sets are classified according to their common functionality and generic system behaviors are specified with method set groupings. Each generic system behavior will be described by a method taxonomy diagram. The refinement of each method taxonomy requires identifying additional constraints on the method sets / contracts as the design of method submodel evolves. The creation of dispatch matchings also takes place in this activity, which requires associating method sets with classes and features. CDSs of the method sets to be implemented should be refined and additional implementation details (i.e., pseudo code) should be documented as needed. The following are the steps in this activity:

    - Combine, rearrange, specialize method sets as needed.
    - Develop and refine method taxonomy diagrams.
    - Create and refine dispatch mappings.
    - Refine, update CDSs.

  Changes might result in the iteration to the preceding activities.

- Develop and refine algorithmic decomposition.

  Client diagrams should be developed for each method set specified in the previous activity. Client diagrams, as used to illustrate the algorithmic or functional decomposition for routines, are crucial to the implementation of the method sets that they are associated with. For example, the designs of service requesting (message passing) or dynamic binding (Section 6.5) are most likely specified in terms of client diagrams. Again, changes might result in revisiting the previous activities.

- Develop and validate instantiations.

  Instantiation diagrams are used to validate the design. Instances specified in the diagrams might be the objects to be created in the implementation (programs). The creation of instantiation diagrams will also aid programmers in the implementation process. However, the validation activity might result in the modification of type diagrams, or class definitions / CIDSs and inheritance diagrams, the revision to the previous activities is therefore required.

5.4 IDEF4/C++ Design Development Process with Multiple Developers

Thus far, we have presented the transformation from IDEF4 to IDEF4/C++ and the activities in the IDEF4/C++ design procedure. However, the development of large-scaled systems often requires multiple developers / teams. As the design evolves, different design versions might be created concurrently and the communication between developers or the development teams tends to be more complicate. This will require a broader view for the identification of the inter-developer or inter-team activities. Consideration of configuration management for the development process and the control for the changes of versions are also required. In the following discussion, we will present an IDEF3 dynamic model describing the system development process with multiple developers / development teams from the perspective of configuration management.

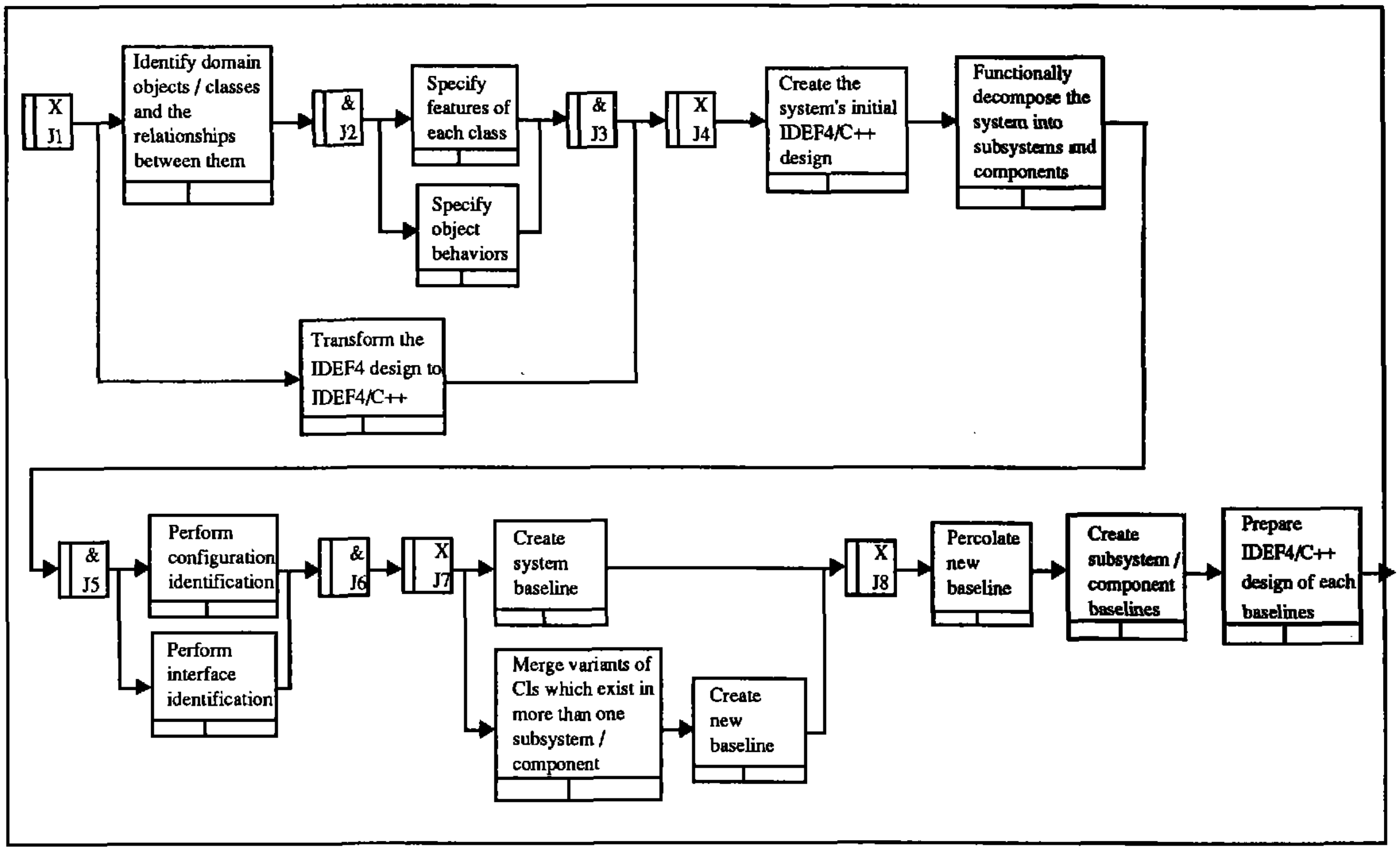
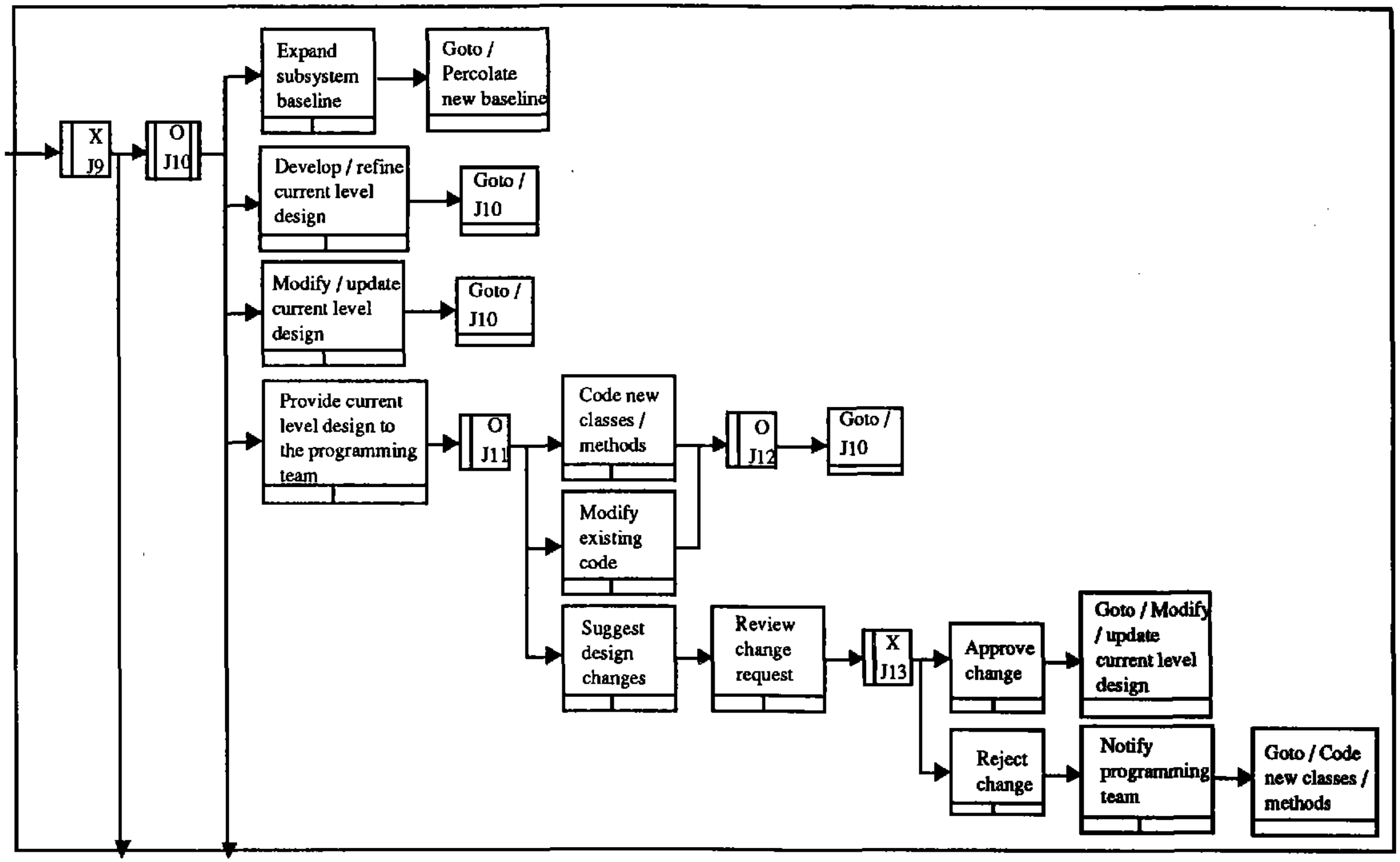
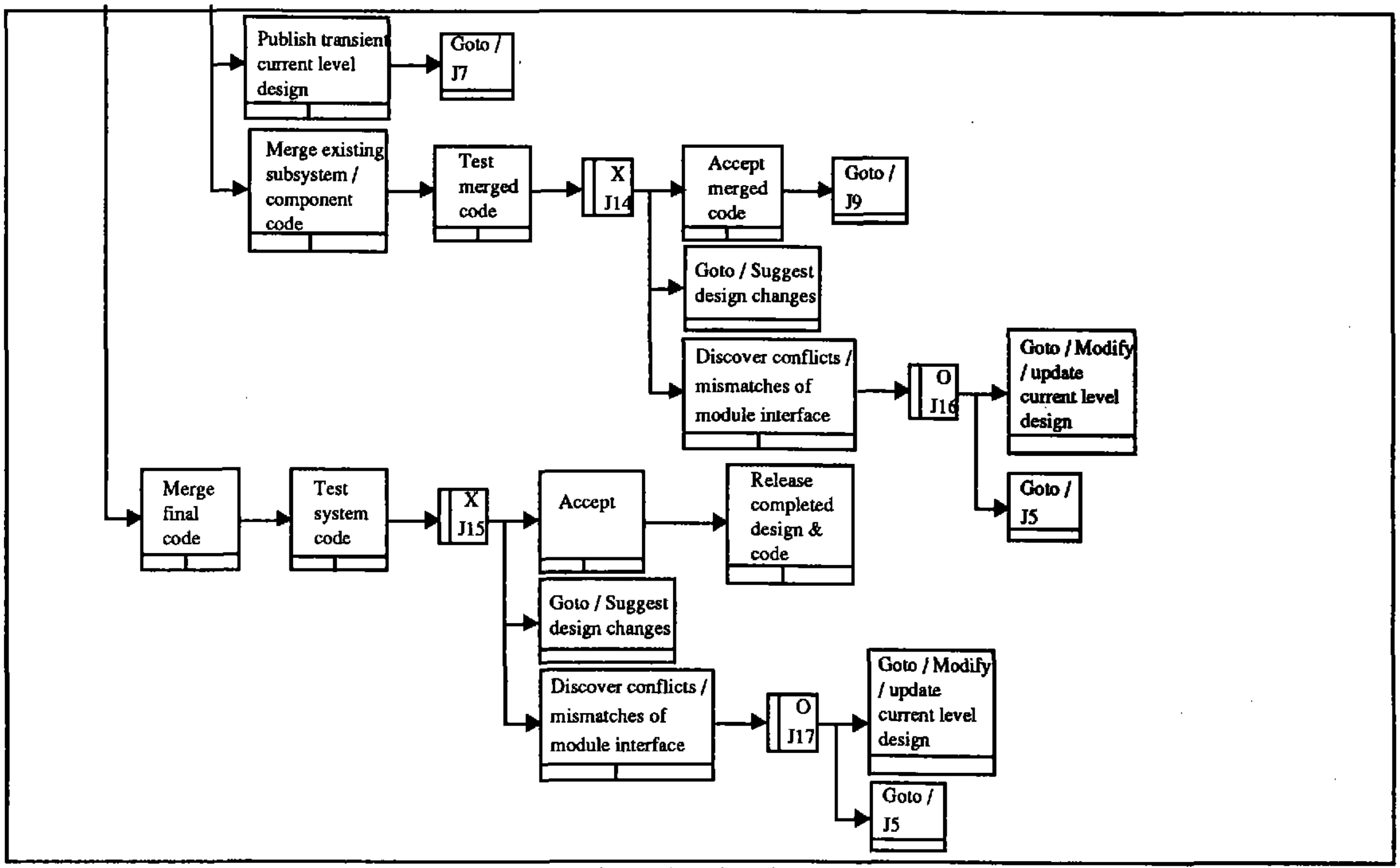

X
J1
Identify domain objects / classes and the relationships between them
&
J2
Specify features of each class
Specify object behaviors
&
J3
X
J4
Create the system's initial IDEF4/C++ design
Functionally decompose the system into subsystems and components
Transform the IDEF4 design to IDEF4/C++
&
J5
Perform configuration identification
Perform interface identification
&
J6
X
J7
Create system baseline
Merge variants of CIs which exist in more than one subsystem / component
Create new baseline
X
J8
Percolate new baseline
Create subsystem / component baselines
Prepare IDEF4/C++ design of each baselines

Figure 5-1. IDEF3 Dynamic Model of IDEF4/C++ System Development Process with Configuration Management.



X
J9
O
J10
Expand subsystem baseline
Goto / Percolate new baseline
Develop / refine current level design
Goto / J10
Modify / update current level design
Goto / J10
Provide current level design to the programming team
O
J11
Code new classes / methods
Modify existing code
Suggest design changes
O
J12
Goto / J10
Review change request
X
J13
Approve change
Goto / Modify / update current level design
Reject change
Notify programming team
Goto / Code new classes / methods

Figure 5-1. (continued)



Publish transient current level design
Goto / J7
Merge existing subsystem / component code
Test merged code
X J14
Accept merged code
Goto / J9
Goto / Suggest design changes
Discover conflicts / mismatches of module interface
O J16
Goto / Modify / update current level design
Goto / J5
Merge final code
Test system code
X J15
Accept
Release completed design & code
Goto / Suggest design changes
Discover conflicts / mismatches of module interface
O J17
Goto / Modify / update current level design
Goto / J5

Figure 5-1. (continued)

Figure 5-1 illustrates the IDEF3 model. The following are the definitions of the terminology used in the model.

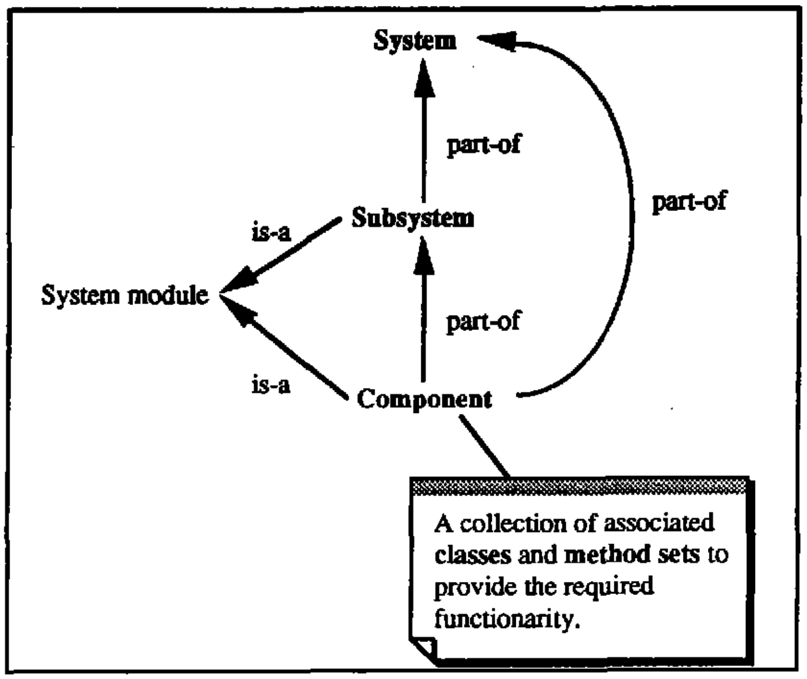

System
part-of
Subsystem
is-a
System module
part-of
part-of
is-a
Component
A collection of associated classes and method sets to provide the required functionarity.

Figure 5-2. Configuration Items defined in IDEF4/C++.

- *Configuration item* (*CI*): System, subsystem, component, class, or method set.

  CI is any design element whose state is to be recorded and whose changes are to be processed (controlled). A CI can be the system, a subsystem, a component, a class, or a method set. Figure 5-2 illustrates the relationships between these CIs. A system can be functionally decomposed into several subsystems and components. Both subsystems and components are system modules. Subsystems can also be decomposed further into subsystems and components, or only components. Components, on the other hand, represent the software modules which require no further decomposition to purchase or build. Any component can be considered as a collection of associated classes and method sets in the context of an IDEF4/C++ design.

- *Configuration identification*: Identifying each CI and its interface.
- *Interface identification*: Identifying all functional characteristics relevant to the interfacing of two or more CIs.
- *Version*: A version is a recorded state of a CI at some point of time.
- *Version tree*: The hierarchy formed by the versions of a CI created over its design evolution.
- *Variant*: A branch of a version tree. Note that the occurrence of variants of a CI indicates that the CI is included in more than one component and the changes to different copies of the CI are made concurrently.
- *Merge*: The process of resolving conflicts between two or more than two variants of a CI and creating a single version for it.
- *Baseline*: A version of a CI serves as a baseline. A baseline represents a well-specified state of a CI in its design evolution. Baselining a CI is the process of assembling the baselines of its element CIs. If the element CIs have variants, baselining a CI will involve the process of merging.
- *Percolate*: The process of decomposing the current baseline of a CI into the baselines of its element CIs.
- *Publish*: The process of preparing the current baseline of a CI to be assembled with other CI baselines at the same level. Publishing a CI results in the re-baselining of the composed CI (the CI at upper level).

The process starts at the creation of an initial design. An IDEF4/C++ initial design can be derived from either (1) specifying / discovering domain objects / classes and the features / behaviors of these objects, or (2) transforming an existing IDEF4 generic design (Section 5.2). Since the design process involves multiple developers / development teams, a functional decomposition for the initial design system, followed up by the activities of configuration identification and interface identification, is therefore necessary. Each decomposed subsystem / component will be assigned to the responsible developers / development teams and the major focus of the design process will be on these system modules.

Before the module design process begins, the system baseline has to be specified and percolated to create the baselines for each modules at subsequent levels. This process will repeat until the percolation reaches the bottom level of the system configuration - where all the modules are components. The developers / development teams therefore base on their own module baselines, carrying out the design activities concurrently. This stage includes activities such as (1) developing and refining the current module, which involves the design activities as described in Section 5.3, or (2) expanding current module, which iterates the process of percolation and creates another level of baselines.

As the design evolves, the working design version might be provided for prototyping or a matured design version might be provided for coding. These activities will revolve around the interactions between design and programming teams if there is a change to the design version which is requested or suggested by the implementors. The approved change request will invoke the process of modifying the existing design version and allow an opportunity for another iteration of team interactions. The working version can also be published to the upper level as needed. However, the decision of publishing a working version will call up the process of re-baselining the upper level module.

If variant versions are created across those components / classes / methods which exist in more than one subsystem / component, the process of merging variants is therefore required before creating the new baseline. The code merging might also reveal some interface conflicts / mismatches between system modules.

The modification at the merged module level or even the process of re-performing interface identification is therefore needed. This brings out another development iteration until the final system reaches its release version. Figure 5-3 presents the basic flow of control of the configuration management in IDEF4/C++.

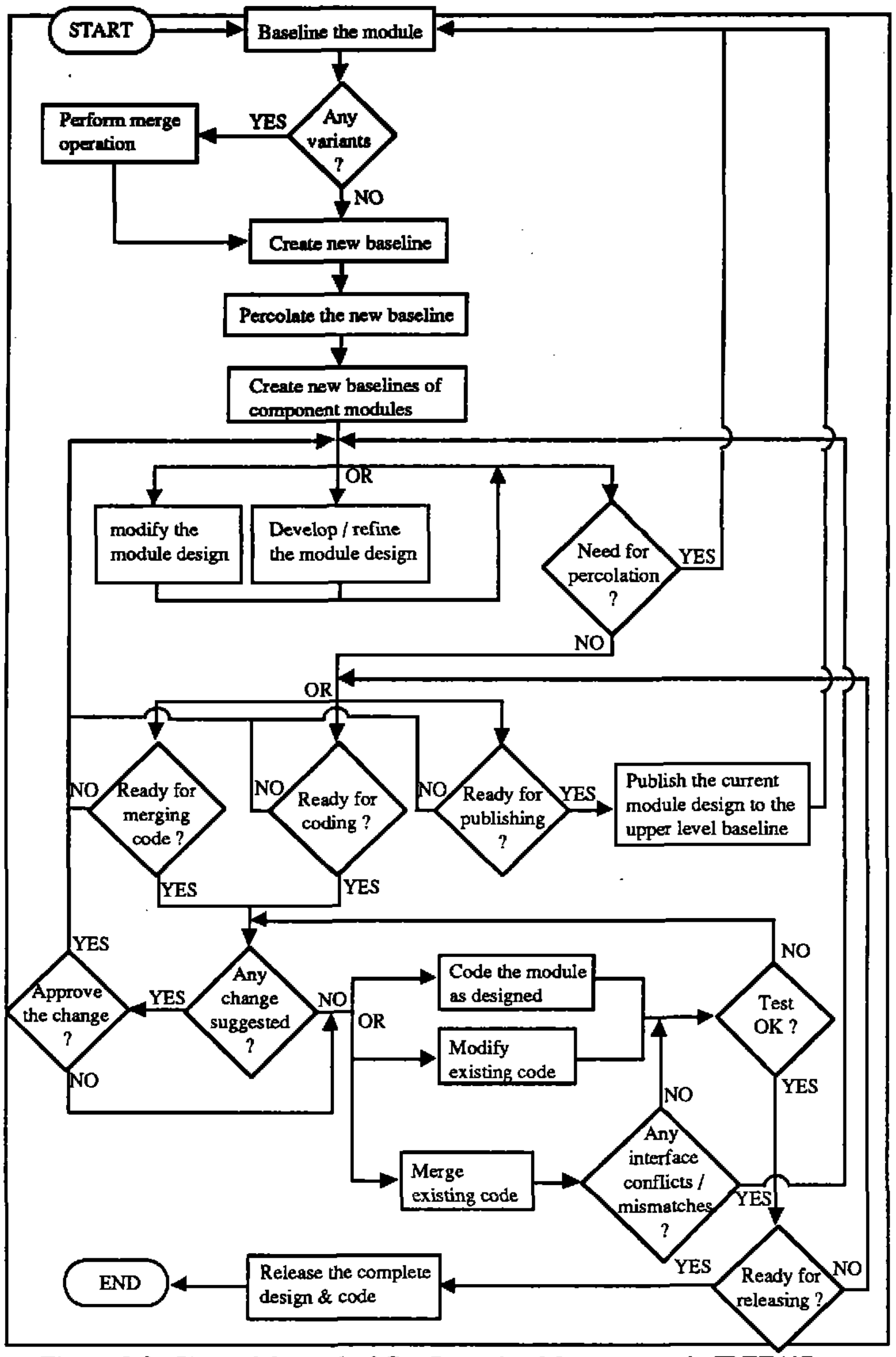


Figure 5-3. Flow of Control of Configuration Management in IDEF4/C++.

CHAPTER VI

METHOD USES

6.1 Introduction

In a sense, object-oriented design is a type of design fashion using the technique of indirection (i.e., abstraction, inheritance, encapsulation, or polymorphism etc.). A good object-oriented designer should be able to practice these techniques competently. Two points are worth emphasizing: the importance of language support for these techniques and the design principles behind them. This chapter focuses on the discussion of these techniques by targeting on the practice of designing with reuse using IDEF4/C++. Several language-dependent design issues such as inheritance and feature access control, constructors and destructors, the design of dynamic polymorphism in IDEF4/C++, and abstract base classes etc., will be addressed prior to the introduction of the techniques for design with reuse. A general set of "rules of thumb", which facilitates the application of reuse issues examined in this chapter, will also be presented.

6.1.1 Inheritance vs. Aggregation

The solution model is not only constructed by the individual objects (classes) distilled from the problem domain, but also those relationships among them as well. Two of the most common relationships captured in the object model are subtyping (IS-A relationship) and containment (HAS-A relationship). Subtyping is implemented by inheritance and containment is implemented by aggregation.

Typically, software components (classes) are reused through the mechanisms of inheritance and aggregation. A new class can be constructed by specializing existing (reusable) base classes or containing the reusable classes as return types of the new class's attributes in order to reuse their functionalities. However, due to the reuse intent, it is possible to apply either of these different mechanisms in the same context (semantics)[12]. Consider the example illustrated in Figure 6-1, which shows two design alternatives for constructing the *Salary-Employee* class. *Salary-Employee* is defined by combining the functionalities of *Employee* and *Salary*. This can be done by 1) defining *Salary-Employee* as a specialization of both *Employee* and *Salary* in the fashion of multiple inheritance, or 2) subclassing *Salary-Employee* from *Employee*, while aggregating *Salary* into *Salary-Employee* class definition. In the first approach, class *Employee* and *Salary* are reused as mixins for providing functionalities by inheritance. Whereas in the second approach, *Salary* is incorporated into *Salary-Employee* as its attribute and functionality is provided through forwarding. The tradeoffs that need to be taken into account between these two alternatives in the same semantics are sometimes subtle and difficult. Final decision usually accompany comprises. However, unless the relationship is exactly a subtyping (generalization / specialization) relationship, aggregation combined with forwarding is preferable for serving the reuse purpose. Section 6.9 will discuss this issue in more details.

[12] If we only focus on directly mapping those nature relationships into our model in the design process, life will be much easier.

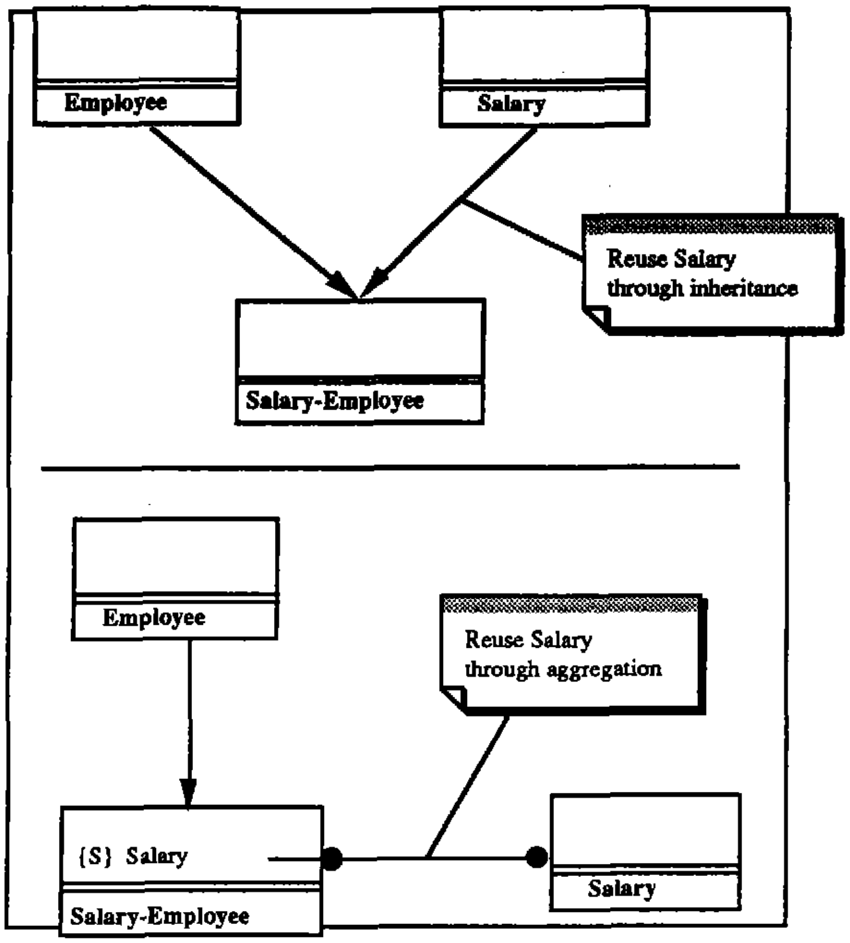


Figure 6-1. Inheritance vs. Aggregation.

6.1.2 Broadening the Design Scope

Typically, the orientation of a design process tends to create classes suitable for use in a particular problem, but are not general enough for broad reuse. The intent of design reuse is the main reason we want to broaden the design scope. It is important to leverage our perspective beyond the current design scope while we are conducting a design process. A broader scope (purpose) for constructing the abstractions (classes/method sets) leads to improved chances for reusing these abstractions in the future. More specifically, broadening our design scope prepares us to deal with new abstractions which are similar to those in the current problem domain. Broadening the design scope also helps to smooth the evolutionary process when the inheritance mechanism is adopted as a means of reuse. A broad design can better accommodate the variety of the intended behaviors than a design from a single perspective can. Generally, the process of broadening the design scope leads to two extremes in terms of the size of abstractions (classes); fat base class and skinny base class. The "fat class" is a consequence of the intent that we want the interface of a base class to be able to provide all possible behaviors for further inheritance purposes. Abstract base classes (mixins) are the examples. On the other hand, the "skinny class" promises that it is generally easily specialized from while specific behaviors can be easily added into the derived classes. However, there are always some trade-offs. The question that we must ask ourselves is: "When broadening our design scope, what is the proper granularity for the size of those classes to be reused?"

6.2 Inheritance and Feature Access Control

As a general rule, classes should avoid exporting their internal structure, even to their derived classes; inheritance is not a license to violate encapsulation. However, when we are concerned about the issue of encapsulation (information hiding), we must also examine the effects of inheritance on the access of features.

In this subsection, we will discuss the language-dependent design issues concerning different types of inheritance and feature access control. In IDEF4/C++, feature access types can be public, protected, or private. Public features are visible to the whole system. Protected features can only be accessed by the owner class and all its directed and undirected derived classes. Private access prevents the features from being accessed by other classes, thereby providing a means of encapsulation. Types of inheritance can be public, protected, or private. In general, a public inheritance provides a means for specifying a subtyping relationship between a base class and its derived class, whereas the private inheritance supports reuse (see Section 6.2.2 and Section 6.9.2). In the following, we will revise the issues of the protected feature access type, private inheritance with access specifier, as well as changing feature access type in the public inheritance.

6.2.1 Protected Feature Access Control

Protected feature access control in IDEF4/C++ is the same as the private access in IDEF4. All the derived classes have the access to those features that are declared as protected in the base class. Figure 6-2 illustrates this notion; so that both derived classes B and C have access to the protected feature f declared in the base class A. Note that the private feature g in class A still can not be accessed by either B or C: consequently even the derived classes have no more right to violate the base class encapsulation than any other class. However, declaring a base class feature "protected", is similar to declaring all the derived classes friends to the base class, which provides an alternative avenue to break the base class's encapsulation. The protected feature access control can be very useful to the design of mixins (abstract classes) so it can be used to prevent these abstract classes from creating instances of their own. This is discussed in Section 6.3.1.

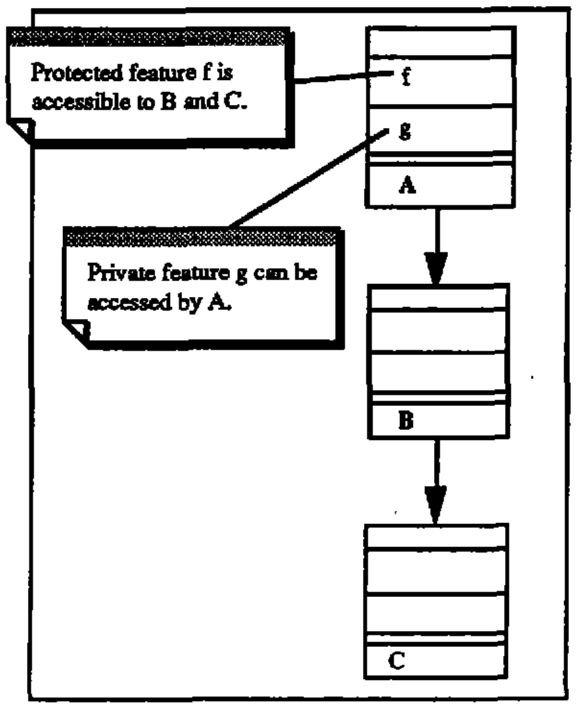

Protected feature f is accessible to B and C.
f
g
A
Private feature g can be accessed by A.
B
C

Figure 6-2. Protected Feature Access Control.

6.2.2 Private Inheritance and Access Specifier

As described in Section 4.3.2, private inheritance collects all the non-private features (public and protected) defined in a base class and redefines their feature access types as private in the derived class. However, by specifying an *access specifier* (Coplien 92) for each inherited feature separately, the access of the inherited feature can be re-specified back to public or protected as needed. Generally, the intent of using a private inheritance is for reuse without breaking any natural relationship between two classes. By redefining the non-private features of a base class as private to a derived class, the functionality that the base class provides can thus be reused by the derived class. Note that the intent of using a private inheritance is different from the public inheritance, which is used mainly to denote a subtyping relationship. However, for some inherited features which are also suitable for constructing the interface of the derived class in a private inheritance, it is necessary to redefine these features (back) to public. For example, consider a *List* class and a *Set* class both illustrated in Figure 6-3. We construct the *Set* class by reusing the class *List* in the fashion of a private inheritance since their functionalities are similar, however, they do not possess a subtyping relationship (*Set* is not a subtype of *List*). As illustrated in Figure 6-3, class *List* has five features; ***head***, ***tail***, ***count***, ***has-item***, and ***insert***. Features ***head*** and ***tail*** have no meaning to a set and are hidden within the private area of class *Set*. ***Insert*** and ***has-item*** providing functionalities that are appropriate to the interface of class *Set*, are also re-shown in the public area of *Set* class box, indicating that their access type is redefined (back) to public. Note that since there is no change to their contracts (implementation constraints), neither symbol "+" nor "!" need to be added to their definitions. Feature *insert* follows the same intent, except that its contracts need to be redefined since no duplicate items are allowed to exist in a set. However, the CIDS of class *Set* should document all the design intents. The C++ implementation is shown as follows[13]:

```
class List {
public:
 void *head();
 void *tail();
 int count();
 Boolean has-item(void*);
 void insert(void*);
};

class Set : private List {
public:
```

[13] This example is extended from James O. Coplien, 'Advanced C++'(p 100).

```
    void insert(void *m);   // redefine its contracts
    List::count;                    // access specifier
    List::has-item          // access specifier
  };
```

Where members *count* and *has-item* only have their names shown in the *Set* class definition, prefixed by the *List* class specifier "List::";

```
List::count;
List::has-item;
```

They are called *access specifiers*. Note that one can not do the opposite. That is, using access specifiers in a public inheritance to change feature access will cause a compiling error in C++. The following section discusses public inheritance.

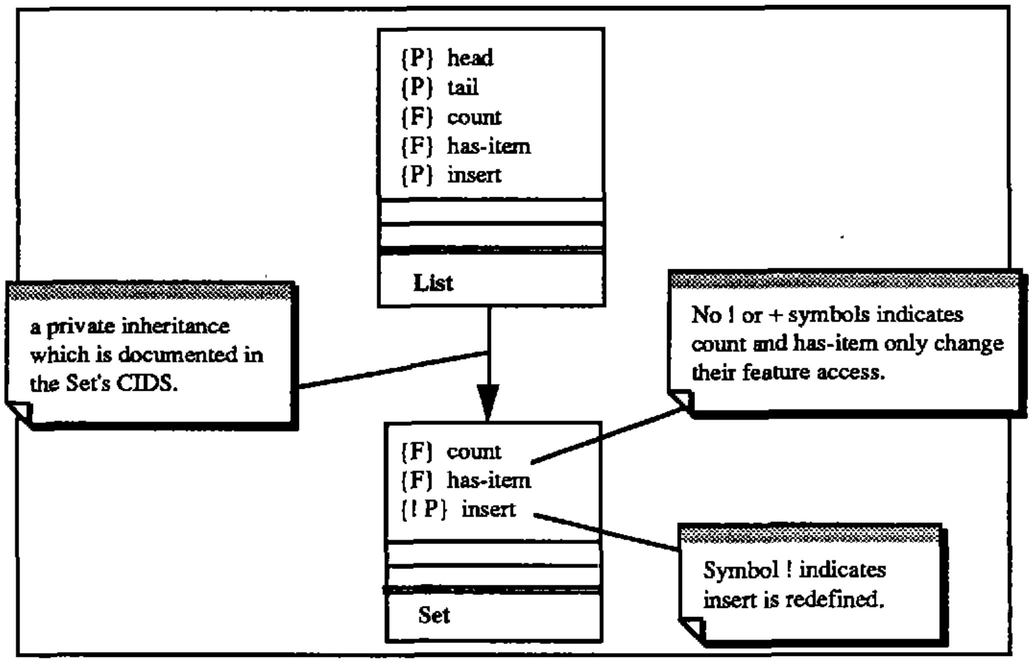

{P} head
{P} tail
{F} count
{F} has-item
{P} insert
List
a private inheritance which is documented in the Set's CIDS.
No ! or + symbols indicates count and has-item only change their feature access.
{F} count
{F} has-item
{! P} insert
Symbol ! indicates insert is redefined.
Set

Figure 6-3. Private Inheritance and Feature Access Control.

6.2.3 Public Inheritance and Feature Access Control

Changing feature access type in a public inheritance is straight-forward. Inherited features which change their feature access in a derived class are only needed to re-display in the intended access control area of the class box. For example, consider a design illustrated in Figure 6-4 where two features are defined in the *Base* class: *method-1* and *method-2*. *Method-2* is virtual to *Derived*, indicating that *Derived* follows all the contracts the inherited feature carries from *Base* and has no intent to redefine it. Two copies of *method-1* are re-displayed in *Derived*; one being public and the other being protected. The public redefined *method-1* in *Derived* is prefixed by a redefining symbol "!", indicating that it has new contracts different from those inherited (for example, adding an *int* parameter). The new contracts can be referred to in the CDS of this copy of *method-1*. The protected redefined *method-1*, on the other hand, has no prefixing redefining symbols. It is re-displayed only for the purpose of showing the change of its access type.

```
class Base {
public:
 void method-1(void);
 void method-2(void);
};

class Derived : public Base {
public:
 void method-1(int);
protected:
 void method-1(void); // change in access type.
};
```

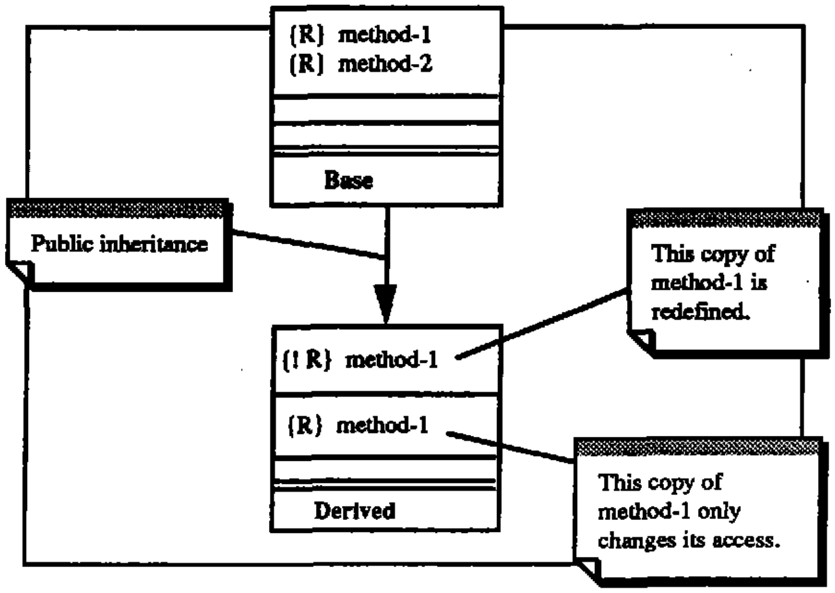

{R} method-1
{R} method-2
Base
Public inheritance
This copy of method-1 is redefined.
{! R} method-1
{R} method-1
Derived
This copy of method-1 only changes its access.

Figure 6-4. Public Inheritance and Feature Access Control.

In IDEF4/C++, a derived class may redefine a feature inherited from a base class. If the intent is to add some new contracts (adding an *int* parameter as in the example), symbols "!" or "+" have to be presented. If the intent is only to change the feature's access, only the name needs to be displayed in the intended access control area.

6.3 Constructors and Destructors

Constructors and Destructors in IDEF4/C++ are used for the instantiation and termination of instances of a class. In this section, we will discuss constructors and destructors in the context of inheritance.

6.3.1 Protected Constructors

Some base classes (such as mixins) are created only for the inheritance purpose. No instances of the mixin classes will be created. One of the design techniques to serve this purpose utilizes protected constructors. For example, consider the design of an employee system; *Employee* class is defined as an abstract base class for the derived classes: *Programmer*, *Analyst*, and *Designer*. Instances exist in the system which are either programmers, analysts, or designers; no instance of a generic employee is allowed. To prevent the creation (incident or on purpose) of instances of *Employee*, we hide the constructors of *Employee* in the protected area. This is demonstrated in the following code:

```
class Employee {
public:
     ................
protected:
  Employee();
private:
     ................
};

class Programmer : public Employee { .......... };
class Analyst : public Employee { .......... };
class Designer : public Employee { .......... };
```

This guarantees that no generic employees will exist in the system since no classes have the access to the *Employee*'s constructor, except for its derived classes. Moreover, by using protected constructors, the base class constructors can still be implicitly called when an instance of the derived class is instantiated. In IDEF4/C++, this intent is to be documented in the CIDS of the base class.

6.3.2 Passing Parameters to Base Constructors

When instantiating an instance of a derived class, the base class constructors will automatically be invoked in the execution of the derived class constructors. However, they (base constructors) can also be invoked explicitly. Consider a shape system, where the class *Square* is a specialization of the class *Rectangle*. The major distinction between them is that the creation of a square needs only one parameter, the length of a side, whereas the rectangle needs two; both length and width. To reduce unnecessary efforts, one may let the constructor of the class *Square* call the constructor of the class *Rectangle*, instead of re-implementing the whole initiating algorithm. This is shown as follows:

```
class  Square : public Rectangle  {
public:
  Square (Point center, int side) : Rectangle (center, side, side) { };
};
```

To fulfill this design intent, the designer needs to document this constraint into the CDS of *Square*'s constructor, and construct a client diagram as illustrated in Figure 6-5 for it.

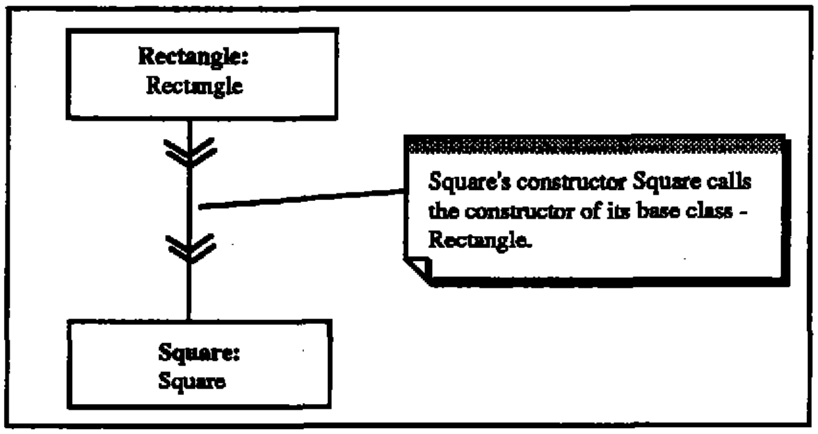

Rectangle:
Rectangle
Square's constructor Square calls the constructor of its base class - Rectangle.
Square:
Square

Figure 6-5. Calling Base Constructors.

6.3.3 Virtual Destructors

Unlike constructors, base destructors cannot be invoked implicitly (automatically) during a cleanup process. Consider the following code example:

```
class Employee {
public:
  ~Employee();
};

Employee *Joe = new Programmer;
  ................
delete Joe;
```

The delete process will have no idea that *Joe* is a programmer since it is typed as *Employee*. Instead of calling the right destructor, *Programmer*'s destructor, the destructor of *Employee* will be invoked. The consequence is that some additional resources allocated for the specialization, *Programmer*, will not be freed by the

Employee's destructor; resulting as garbage in the system. To avoid this, one can declare *Employee*'s destructor *virtual*:

```
class Employee {
public:
 virtual ~Employee();
};
```

If the destructor of a base class is declared virtual, the system will then automatically invoke the proper destructor for its derived types (i.e., *Programmer*) and call the base destructors afterwards. This design intent is to be documented in the CIDS of the base class.

6.4 Pure Virtual Functions and Abstract Base Classes

Recall that, in Section 6.3.1, in order to prevent any incidental creation for the instances of a generic base class (i.e., *Employee*), we declare the base constructors as protected. Another design alternative is to use pure virtual functions. In IDEF4/C++, pure virtual function declaration is specified by using the feature symbol "V0", indicating the feature (a computation-initiating feature) is not intended to have its function body. This enforces an obligation on the derived class to redefine / override the pure virtual functions and prevents any instantiation of the base class. Note that an abstract base class is a specialization of an abstract data type (ADT), if all its member functions are pure virtual. This is one of the techniques of design with reuse which will be addressed more specifically in Section 6.9.1.

6.5 Designing Dynamic Polymorphism in IDEF4/C++

In general, the object-oriented technique is characterized by inheritance and run-time binding (dynamic polymorphism). Inheritance provides a hierarchical structure for defining generalization / specialization relationships between classes. Attributes common to several classes can be moved up to their base classes (generalization) and derived classes can specify their own behaviors by redefining those more general behaviors in the base classes (specialization). Run-time binding encapsulates these behavioral details in the inheritance hierarchy and simplifies the implementation of the use of them in the program. Inheritance together with run-time binding organize a design in a way of supporting software reuse. For example, consider a class hierarchy with a base class *Shape* and its two direct derived classes, *Circle* and *Rectangle*, and each class has a behavioral feature, *draw*. *Draw* features in *Circle* and *Rectangle* are redefined from the more general *draw* in the base class *Shape* through the inheritance mechanism. To draw a shape object in the system, a generic *draw* function is called which is augmented with the intended object (it might be known or not at compile-time). A good design (in terms of reusability) indicates that Adding a new class, for instance a *Triangle* class, as a derived class in the system requires no modification to be done for the implementation of the generic *draw* function. In other words, extension or modification for the system should have as minimal an impact on the existing design / code as possible. We address how to achieve this intent in IDEF4/C++ as follows.

In IDEF4/C++, inheritance relationship is specified in the inheritance diagrams, whereas run-time binding is supported in terms of defining those behavioral features as virtual and invoking them through a public base class reference or pointer. Consider the example described previously. The class inheritance hierarchy is illustrated in Figure 6-6 and the C++ class definitions for these classes would look like:

```
class Shape {
public:
 virtual void draw (void);        // defined as a virtual function.
   ........................
};

class Circle : public Shape {
public:
 void draw (void);        // redefined draw.
   .......................
};

class Rectangle : public Shape {
public:
 void draw (void);        // redefined draw.
   ......................
};
```

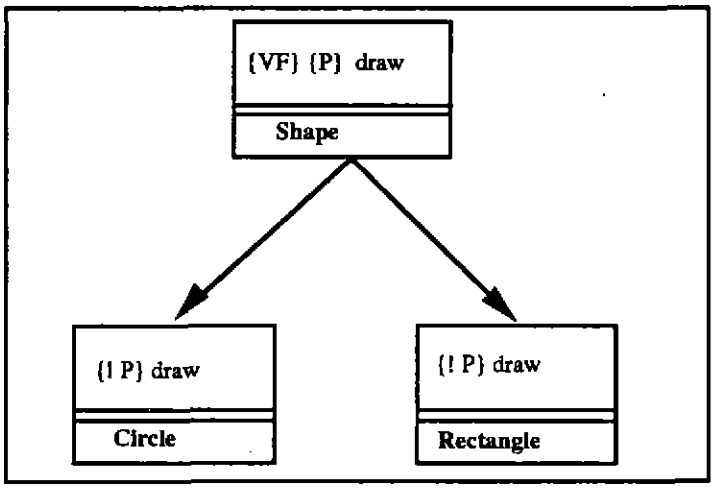


Figure 6-6. Inheritance Diagram for Class Shape.

As shown in both the inheritance diagram and the code example, *draw* of *Shape* is defined as a virtual function. To construct a run-time binding mechanism for the draw behavior, an independent *draw* function is defined and it is augmented with the type *Shape*. It looks like:

```
void draw (Shape &obj)
{
 obj.draw();
}
```

The trick is that instances of the derived classes *Circle* or *Rectangle* are also instances of the base class *Shape*[14]. The type of the instance that the parameter *obj* references (might be *Circle* or *Rectangle*) will be resolved at run time to invoke the "right" draw method.

```
Circle          circle;
Rectangle       rectangle;
Shape &s = circle;  //s is a circle.
draw (s);           // draw a circle.
s = rectangle;  // s is a rectangle.
draw (s);           // draw a rectangle.
```

The independent draw function need not know about any future evolution of *Shape* hierarchy (for example, a new *Triangle* class), as long as the intended object holds a redefined *draw* inherited from the *Shape* class. The inheritance diagram and method taxonomy diagram with dispatch mappings of the example are illustrated in Figure 6-7(a) and (b). Where the generic *draw* function does not have a dispatch mapping to any class in the design, one should indicate that it is independent and

[14] The substitution property of objects was discussed in Section 3.2.1.

does not belong to any class. Figure 6-7(c) illustrates the client diagram for this generic *draw*, showing that the generic *draw* function calls the *draw* function defined in class *Shape*. Since *Shape* defines its *draw* as a virtual function, the actual supplier will not be invoked until run-time. However, the CDS for the generic *draw* should document all the intent.

This run-time type resolution encapsulates the implementation details in inheritance hierarchy from the program. In turn, it simplifies the extension of class hierarchy. Adding a new derived class, "*Triangle*", is straight-forward and will not involve any modification of existing code about the generic *draw*.

```
Triangle       triangle;
s = triangle;  // s is a reference to Triangle.
draw (s);      // draw a triangle.
```

In IDEF4/C++, in order to add the *Triangle* class into the system, we only need to modify the inheritance diagram and the method taxonomy diagram (Figure 6-8(a) and (b)). No further change is required for the client diagram of the generic *draw* (same as Figure 6-7(c)). Easier extension of the design and the reusability of the software component is therefore gained.

Note that, the inheritance links between all the derived classes and their base class have to be public to support the run-time type resolution. Protected and private inheritance links do not have the provision for this implicit type conversion.

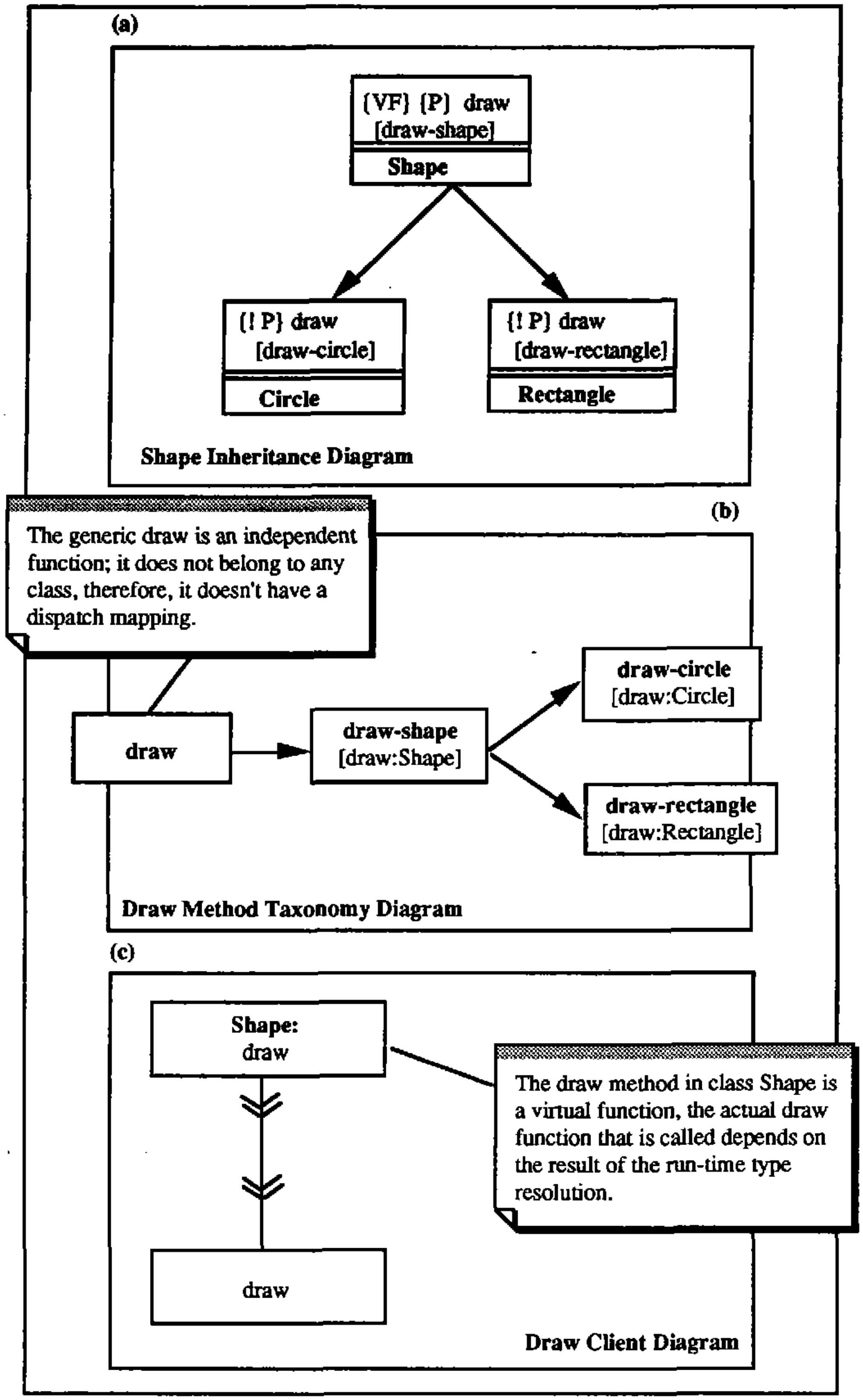

(a)
{VF} {P} draw [draw-shape]
Shape
{! P} draw [draw-circle]
Circle
{! P} draw [draw-rectangle]
Rectangle
Shape Inheritance Diagram
(b)
The generic draw is an independent function; it does not belong to any class, therefore, it doesn't have a dispatch mapping.
draw
draw-shape [draw:Shape]
draw-circle [draw:Circle]
draw-rectangle [draw:Rectangle]
Draw Method Taxonomy Diagram
(c)
Shape: draw
The draw method in class Shape is a virtual function, the actual draw function that is called depends on the result of the run-time type resolution.
draw
Draw Client Diagram

Figure 6-7. IDEF/C++ Design for Run-time Binding.

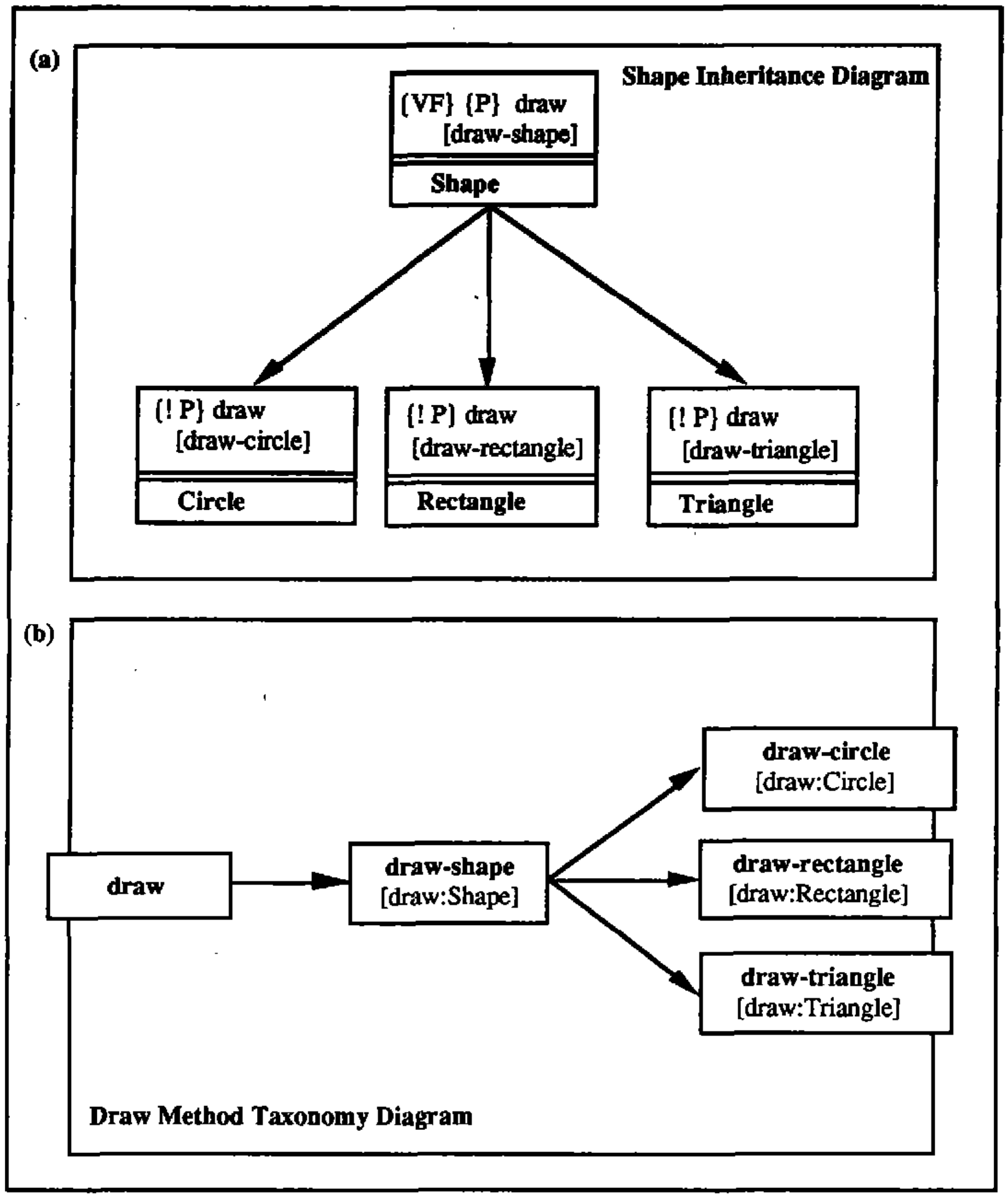

(a)
Shape Inheritance Diagram
{VF} {P} draw [draw-shape]
Shape
{! P} draw [draw-circle]
Circle
{! P} draw [draw-rectangle]
Rectangle
{! P} draw [draw-triangle]
Triangle
(b)
draw
draw-shape [draw:Shape]
draw-circle [draw:Circle]
draw-rectangle [draw:Rectangle]
draw-triangle [draw:Triangle]
Draw Method Taxonomy Diagram

Figure 6-8. Adding a New Class - Triangle.

In summary, the design discussed in this subsection provides a simple approach for implementing dynamic polymorphism in IDEF4/C++. It simplifies the software maintenance process because the modification of class hierarchy has the least impact on existing code, providing a means for software reuse.

6.6 Implementation for Conflicting and Non-conflicting Features

In an IDEF/C++ model, generalization / specialization relationships between classes are specified in inheritance diagrams; whereas between behaviors, it is specified in method taxonomy diagrams. Specialization between a base class and a derived class is actually implied by the set of specialized features inherited from the base class but redefined in the derived class. These redefined features are mapped to their associated method sets which carry the same semantics. In IDEF4/C++, the semantics of "specialization" can be categorized into two different types of additional constraints: conflicting and non-conflicting, represented by symbols "!" and "+", respectively. Conflicting constraints are those that specialize or redefine the inherited constraints. As a result, the redefined feature overrides / shadows the inherited one. Non-conflicting constraints, on the other hand, merely represent the addition of constraints that are new and independent to the inherited constraints. This indicates that the implementation of the redefined feature will also execute the implementation of its inherited feature in order to fulfill the unchanged constraints. For example, consider the following two different C++ implementations of the redefined feature - *method-1*,

```
class Base {
public:
 void method-1 (parameter-1);
    ..................................
};

class Derived : public Base {
public:
 void method-1 (parameter-1);
    ..................................
};


(1)
void  Derived::method-1 (parameter-1 p1)
{ ........... };


(2)
void  Derived::method-1 (parameter-1 p1)
{
   ..................
  Base::method1(p1);     // call Base's method1.
   ..................
};
```

In the first implementation, the feature *method-1* redefined in *Derived* has its own (new) implementation (it does not call *Base*'s *method-1*), indicating that the redefined constraints held by this copy of *method-1* are conflicting to the inherited constraints of the *method-1* in *Base* and that the *method-1* in the *Derived* class shadows the *method-1* in the *Base* class. In the second implementation, the *method-1* in *Derived* calls the one defined in *Base*, indicating that the constraints are non-conflicting and that *Derived*'s *method-1* has to execute *Base*'s *method-1* to fulfill those inherited constraints. Figure 6-9 illustrates the intended IDEF4/C++ design for each implementation, where symbol "!" clearly specifies the conflicting situation and symbol "+" specifies the non-conflicting additional constraints.

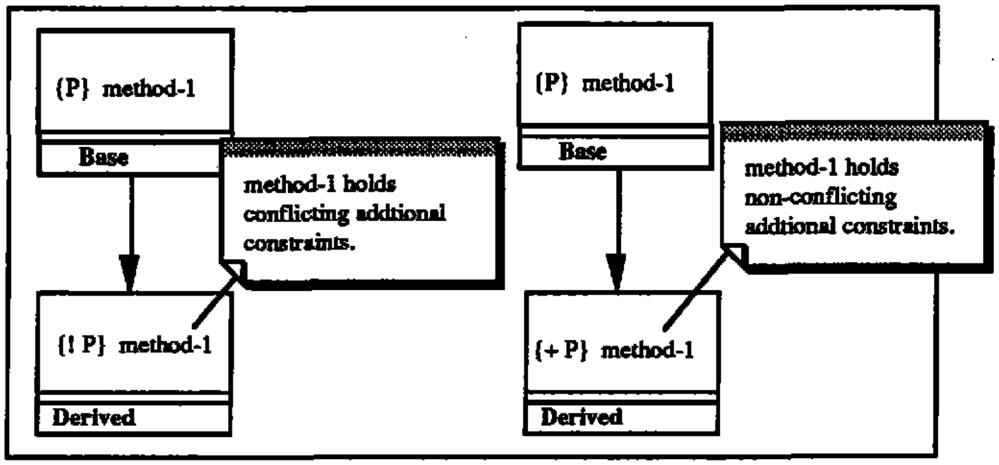

{P} method-1
Base
method-1 holds conflicting addtional constraints.
{! P} method-1
Derived
{P} method-1
Base
method-1 holds non-conflicting addtional constraints.
{+ P} method-1
Derived

Figure 6-9. Conflicting and Non-conflicting Constraints.

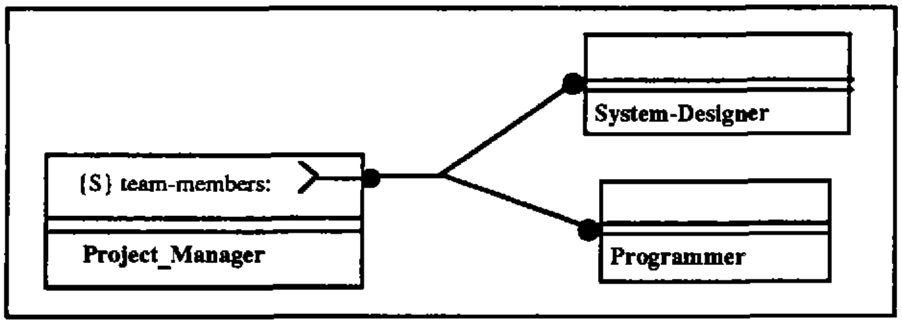


Figure 6-10. Example of Not Allowable Type Diagrams.

6.7 Features with Multiple Return Types

Design of multiple return types is not explicitly supported in the IDEF4/C++ type diagrams; that is, every feature described in a type diagram can have one and only one type link fanning out from its defining class box to its return type. A design such as the one described in Figure 6-10 is not allowed. However, in reality, for those collection-type features such as array, list, or queue etc., elements in the collection might have different return types (but similar in some sense). An approach to solve this problem is to define a general base class for all these similar but specialized types and have the type link of the feature point to the general type.

This is especially important in IDEF4/C++, due to the reason that C++ is a strong-typed language. Consider the example described in Figure 6-10 where class *Project-Manager* has a collection-type feature, *team-members*, which collects the team members of a project. A project development team will consist of members such as system analysts, system designers, and programmers as well. To model this situation, a base class, *Employee*, is introduced as the general base class for the classes *System-Analyst*, *System-Designer*, and *Programmer* . Figure 6-11(a) presents the inheritance diagram. The type diagram for this approach is illustrated in Figure 6-11(b), showing that the type link for *team-members* points to the general type *Employee*. The C++ code implementation for the design will look like:

```
class Project-Manager {
public:
 Employee *team-members[10];
   .......................................
} John;

class Programmer : public Employee { ...... } Tim;
class System-Analyst : public Employee { ...... } Martha;
class System-Designer : public Employee { ...... } Ted;

John.team-members[0] = &Tim;
John.team-members[1] = &Martha;
John.team-members[2] = &Ted;
```

This *Employee* type collection feature, *team-member*, therefore can collect different types of instances, such as programmer *Tim*, system analyst *Martha*, and system designer *Ted* (as shown above). Through C++ *type conversion*, the multiple return types for such a collection feature is therefore feasible. However, implicit type conversion (as in the example) is only supported through public inheritance. For non-public inheritance links such as protected or private, an explicit type casting must be provided in the code.

```
class  System-Analyst : protected Employee  { ...... }  Martha;
class  System-Designer : private Employee  { ...... }  Ted;


John.team-members[1] = (Employee *) &Martha;
John.team-members[2] = (Employee *) &Ted;
```

Again, all these intents and constraints for implementing the design must be explicitly documented in the CIDS for the class *Project-Manager*.

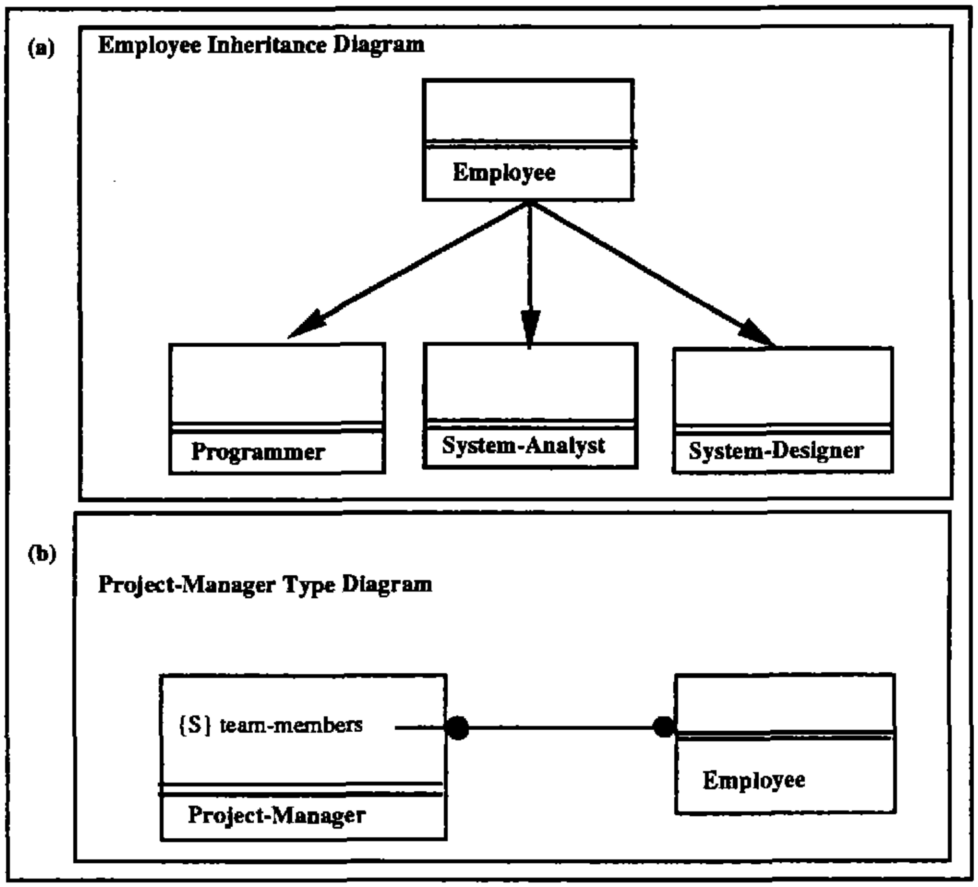


Figure 6-11. Designing Multiple Return Types for Feature Team-members.

6.8 Avoiding Redundancy in Multiple Inheritance

One of the significant problems that might occur in a multiple inheritance structure occurs with name conflicts between inherited features. In Section 4.3.3, we have discussed this problem and introduced the virtual inheritance declaration for resolving the conflict (Figure 4-7). However, resolving name conflicts is considered an important design issue and designers of object-oriented systems should avoid any name conflicts in their designs. In the following section, we will discuss more details of this issue, as well as how to avoid the redundancy accompanying the resolution process of conflicts.

Name conflicts can be categorized into two types: ambiguities (conflicts) between data type features and behavior type features. Recall that the example shown in Figure 4-7, where ambiguity between two copies of the inherited data type feature name in the derived class *Manager* can be avoided by declaring virtual inheritance. For the ambiguities between inherited behaviors, consider the example described in Figure 6-12, where the class *Project-Leader* inherits the features from both *Manager* and *Designer*; indicating that a project leader is also responsible for the system design work.

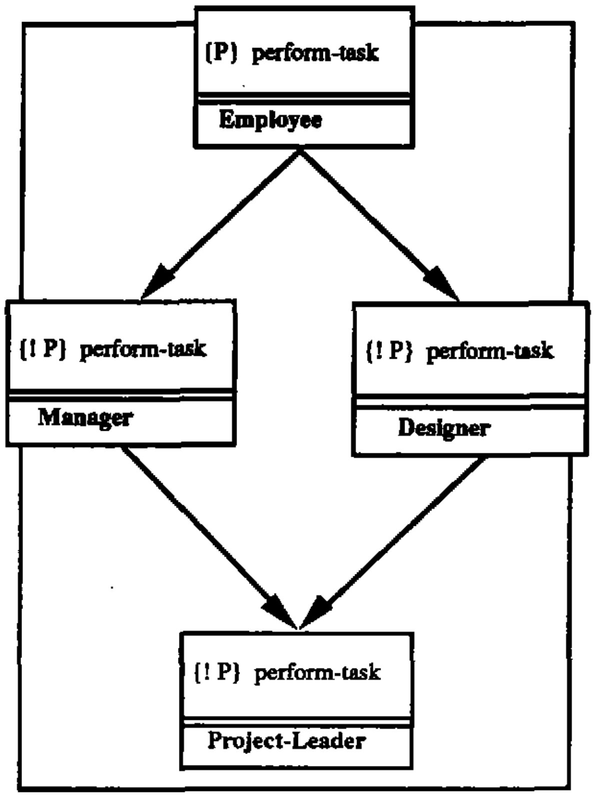


Figure 6-12. Multiple Inheritance of the Class Project-Leader.

Manager and *Designer* both redefine their own behavioral feature - *perform-task*, indicating that they possess different tasks (responsibilities) in the development of a project. The ambiguity occurs because *Project-Leader* inherits both behaviors. The resolution of this conflict is described in the client diagram of the *perform-task* behavior redefined in *Project-Leader* and illustrated in Figure 6-13; where the *Project-Leader*'s *perform-task* calls both *Manager*'s *perform-task* and *Designer*'s *perform-task.*

```
class Employee {
public:
 void perform-task (void);
  .........................  };
```

```
class Manager : public Employee {
public:
 void perform-task (void);        // overriding Employee::perform-task.
   ........................  };


class Designer : public Employee {
public:
 void perform-task (void)         // overriding Employee::perform-task
   ............................  };


class Project-Leader : public Manager, public Designer {
public:
 void perform-task (void)         // redefined for its own behavior.


// The behavior of a project leader to perform his/her tasks can be
// thought of as a combination of the managing and designing work.
void  Project-Leader::perform-task (void) {
        ..........................       // work particular to a project leader
        Manager::perform-task();
        Designer::perform-task();
}
```

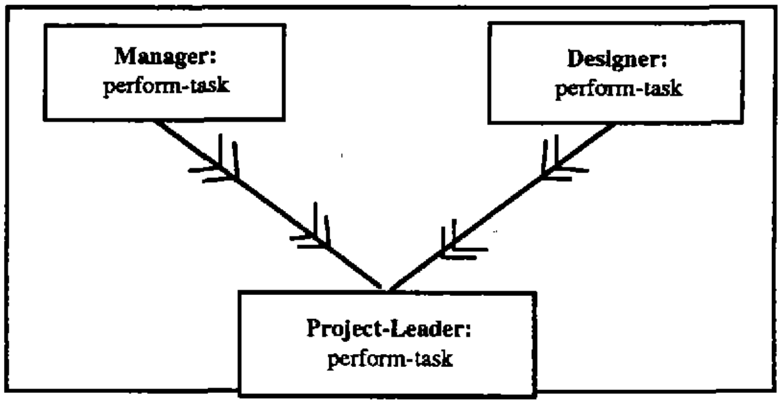


Figure 6-13. Client Diagram of Project-Leader's Perform-task.

The order of performing a manager's tasks and a designer's tasks depends on the domain requirements. However, since both *Manager* and *Designer* inherit this behavior from *Employee*, it is not surprising to see that these traits possess common behavior. Redundancy might occur, but avoided with careful design. First, we separate that common behavior (most likely, it comes from *Employee*) from each of the feature *perform-tasks* possessed in *Manager* and *Designer* and then "re-arrange" the design:

```
class Manager : public Employee {
public:
 void perform-task (void) {
        perform-managing-task(); // perform its own behavior
        Employee::perform-task(); // perform the common behavior
 };
protected:
 void perform-managing-task (void);
  ..............................  };

class Designer : public Employee {
public:
 void perform-task (void) {
        perform-designing-task(); // perform its own behavior
        Employee::perform-task(); // perform the common behavior
 };
protected:
 void perform-designing-task (void);
  .............................  };
```

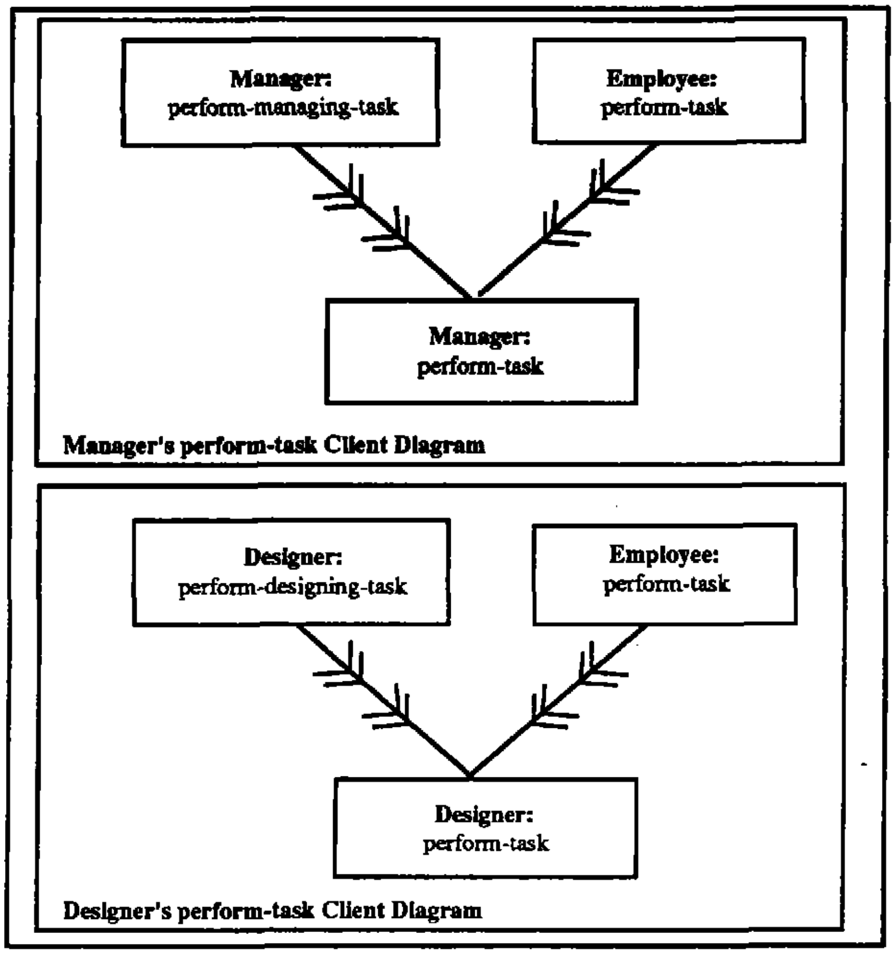


Figure 6-14. Separating Perform-tasks in Manager and Designer.

Figure 6-14 shows the client diagrams. Both the managing and designing tasks are separated and declared as protected behaviors. The same design is applied to the derived class *Project-Leader*. The redundancy in *perform-task* behavior therefore can be avoided and it can also be more selective on ordering these separated activities:

```
class Project-Leader : public Manager, public Designer {
public:
 void perform-task (void) {
        Manager::perform-managing-task();
        Designer::perform-designing-task();
```

```
            perform-project-leading-task();
            Employee::perform-task();
    };
    protected:
     void perform-project-leading-task (void);
       ..................................  };
```

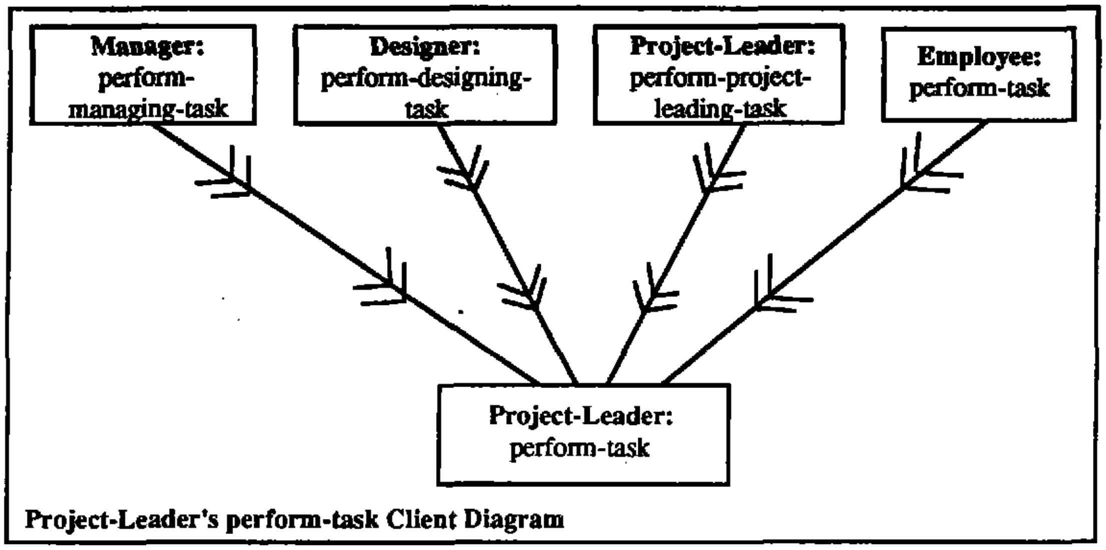

Manager: perform-managing-task
Designer: perform-designing-task
Project-Leader: perform-project-leading-task
Employee: perform-task
Project-Leader: perform-task
Project-Leader's perform-task Client Diagram

Figure 6-15. Avoiding Redundancy in Project-Leader's Perform-task.

Figure 6-15 illustrates the client diagram of *Project-Leader*'s *perform-task* behavior; the redundancy of the common behaviors is eliminated.

6.9 Design with Reuse

In this section, we will address techniques of design with reuse in IDEF4/C++ first. We will then summarize the chapter by giving a set of rules of thumb of this issue.

The techniques that object-oriented design provides: encapsulation, inheritance, and abstraction etc., are sometimes confused with the approaches to design with reuse. One might think a design that makes the most effective use of inheritance promises reuse or a design that follows the direct mapping from the problem domain provides more chances for reuse. This is not necessarily true. An optimized class inheritance structure or a nature object structure do not really promise reusability. Design with reuse is a process that needs careful observation and deep understanding of the problem domain, patience in the process of prototyping and refining, and intelligent application of the techniques for reuse.

In general, reusability can be achieved through two major mechanisms, inheritance and aggregation. However, the techniques discussed in the following section utilize these mechanisms in different flavors.

6.9.1 Abstract Base Class and Pure Virtual Functions

The first technique of reuse is using abstract base classes and pure virtual functions. Base classes possess virtual functions defined to be "pure" (at least one virtual function), indicating that no instances can be created from the class. Therefore the base class is abstract (if such instances exist, the system would not know how to deal with the behavior specified by a pure virtual function since no function body is defined for that function). This is a quite intuitive approach for reuse purpose. Recall that, in the discussion of broadening the design scope for reuse, abstract base classes are usually one of the extreme results that we will get. An abstract base class which is designed for reuse often possesses a "fat" (enlarged) interface due to the reason that we tend to provide a complete set of (pure) virtual functions in order to sufficiently describe all the behaviors. An example of an abstract base class with virtual functions is illustrated in Figure 6-16, showing that the class *Drill-Machine* is designed as an abstract base class for specializing different types of drilling machines. Specializations such as *Gang-Drill-Machine*, *Radial-Drill-Machine*, and *Turret-Drill-Machine*, can be defined through a public inheritance from *Drill-Machine*. Each specialization has to redefine its own behaviors by filling out the bodies of those pure virtual functions. For example, *drill* is defined as a pure virtual function in *Drill-Machine* to represent the drilling behavior. However, this drilling behavior is only a generic behavior; different types of drilling machines have different ways for drilling. Specializations have to override *drill* in order to make their own instances. This enforces the intent of reuse, which is the purpose that we design the class *Drill-Machine*.

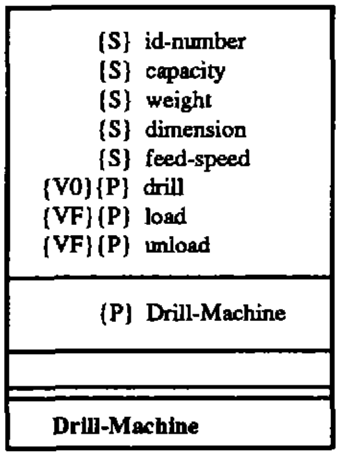


Figure 6-16. An Abstract Base Class - Drill-Machine.

6.9.2 Private Inheritance and Forwarding

In Section 6.2.2, we discussed the design technique for using a private inheritance with access specifiers. This technique can be applied for the reuse purpose as well. In general, a public inheritance captures the nature subtyping (specialization) relationship between a base class and a derived class, while a private inheritance implements the intent of reuse. Through a private inheritance, the functionalities that a base class provides can be reused by the derived class. The base and the derived class do not really need to maintain a subtyping relationship. Consider the example given in Section 6.2.2, class *Set* is not a specialization of class *List*. However, due to our intent to reuse the functionalities (a set of functions) provided by *List* to simplify the design of *Set*, *List* is privately inherited by *Set*. To be able to more efficiently use these inherited functionalities, access specifiers are applied to switch the intended behavioral features from private back to public. Another alternative is using *forwarding*. Figure 6-17 shows how to use the client diagram to design forwarding in this sense. Features *set-count* and *set-has-item* are new defined in the public area of the class *Set*. Instead of changing the feature access type of *count* and *has-item* (both are inherited from *List*), *count* is called by the public feature *set-count* and *has-item* is called by the public feature *set-has-item* in the class *Set*.

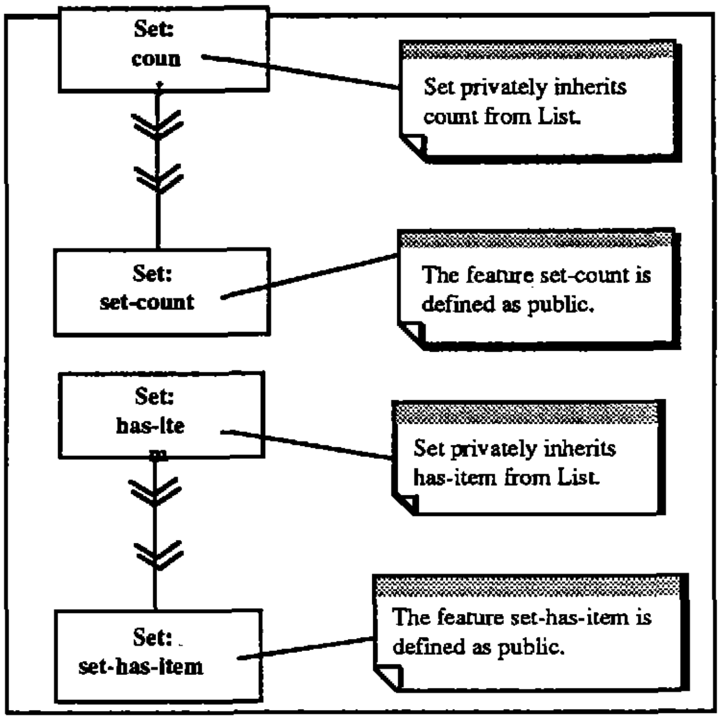

Set:
coun
Set privately inherits count from List.
Set:
set-count
The feature set-count is defined as public.
Set:
has-ite
m
Set privately inherits has-item from List.
Set:
set-has-item
The feature set-has-item is defined as public.

Figure 6-17. Using Forwarding in Private Inheritance.

```
class  Set : private List  {
public:
  void Insert(void *);
  // public feature set-count forwards to the private count.
  Int set-count (void) { return count(); };
  // public feature set-has-item forwards to the private has-item.
  Boolean set-has-item (void*) { return has-item(); };
};
```

6.9.3 Aggregation with Forwarding

As mentioned previously, aggregation is another mechanism that can be used for the reuse purpose. The class to be reused is declared as a type of a private feature defined in the new class. The new class, therefore, can reuse the functionalities provided by that reusable class through forwarding. Reconsider the *List* and *Set* example again. One might feel that inheriting *Set* from *List* confuses the nature semantics between them. It just does not seem right. Figure 6-18 presents the other approach that uses aggregation. The class *List* is contained in the class *Set* and is handled by the feature *alist*. The functionalities of *List* can therefore be reused by forwarding those behavioral features, such as *count* and *has-item*, from *Set* to *List* through the handle *alist*. This forwarding is described by the client diagrams of *count* and *has-item*, which are shown in Figure 6-19 and 6-20.

```
class Set  {
public:
  int count (void) { return alist.count(); };
  Boolean has-item (void *item) { return alist.has-item(item); };
  void insert (void *item) { if (!has-item (item)) alist.insert(item); };
private:
```

```
List alist;
};
```

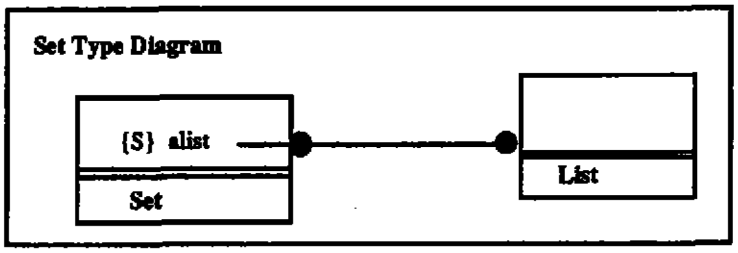


Figure 6-18. Reuse List by Aggregation.

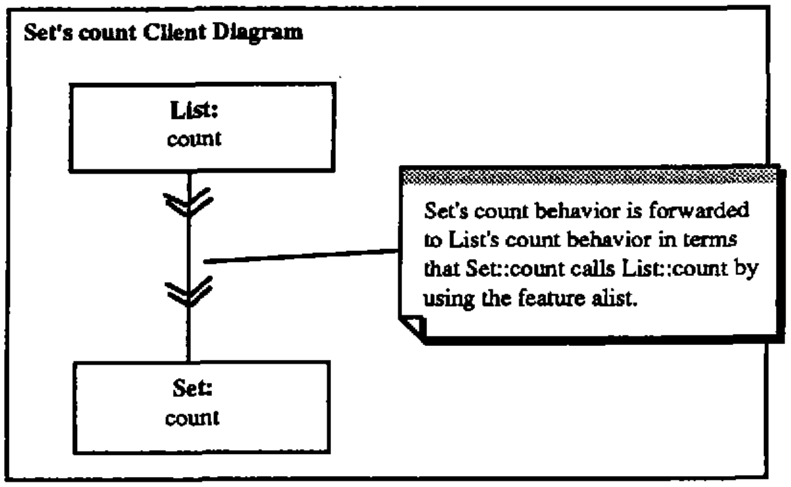

Set's count Client Diagram
List:
count
Set's count behavior is forwarded to List's count behavior in terms that Set::count calls List::count by using the feature alist.
Set:
count

Figure 6-19. Forwarding the Count Behavior.

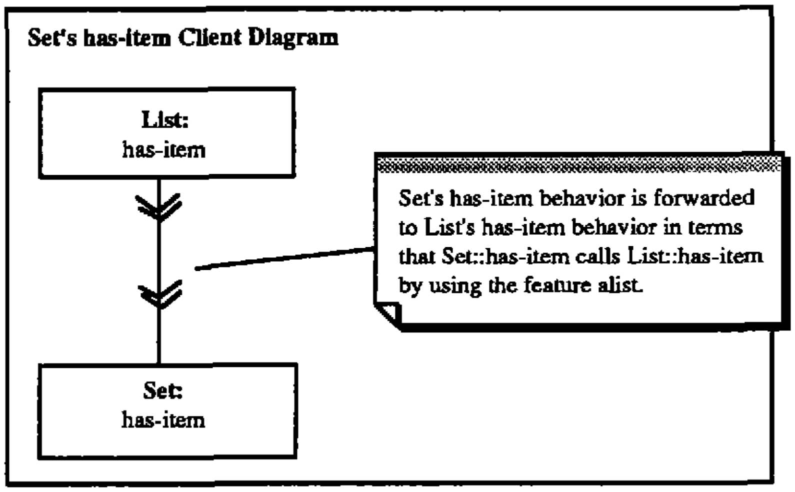

Set's has-item Client Diagram
List:
has-item
Set's has-item behavior is forwarded to List's has-item behavior in terms that Set::has-item calls List::has-item by using the feature alist.
Set:
has-item

Figure 6-20. Forwarding the Has-item Behavior.

By using aggregation with the forwarding technique, we are able to reuse the functionalities of *List* while the semantics of class *Set* can be preserved. Note that reuse by aggregation indicates that no significant relationships exist between two classes. *Set* associates with *List* purely for the purpose of reuse. Reuse by inheritance suggests a chance for violating the encapsulation of the reused base class (its protected features); whereas reuse by aggregation avoids this drawback in a nature way.

6.9.4 Delegation

Another technique of reuse is delegation. Delegation is to object instances what inheritance is to object classes. In delegation, a behavioral feature of an object can be forwarded to another object, invoking the delegated behavior of the second object in the context of the first one. Delegation provides mechanism for two or more separate objects to appear as one. This is particularly helpful in simulating multiple inheritance as the example presented in the following. With language such as Actors and Self support delegation; in the IDEF4/C++, we use forwarding to simulate delegation[15]. Consider a class inheritance hierarchy as illustrated in Figure 6-21. Class *Research-Assistant* inherits both class *Employee* and *Student* so that *Employee* has a behavioral feature - *work* and *Student* has a behavior feature - *study*. Instead of modeling these class relationships as shown in Figure 6-21, in delegation, we construct these classes in the hierarchy as presented in Figure 6-22.

[15] Delegation can be more precisely simulated by overloading the pointing "->" operator. See C++ language reference for more details.

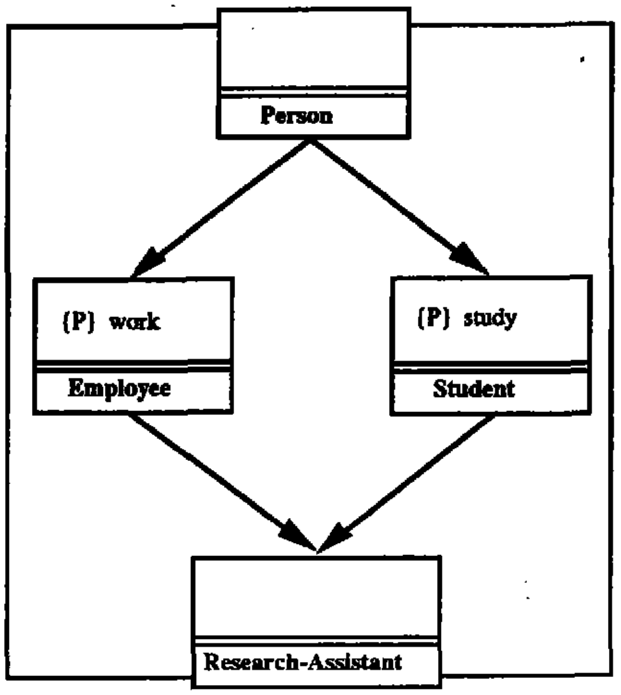


Figure 6-21. Multiple Inheritance of Class Research-Assistant.

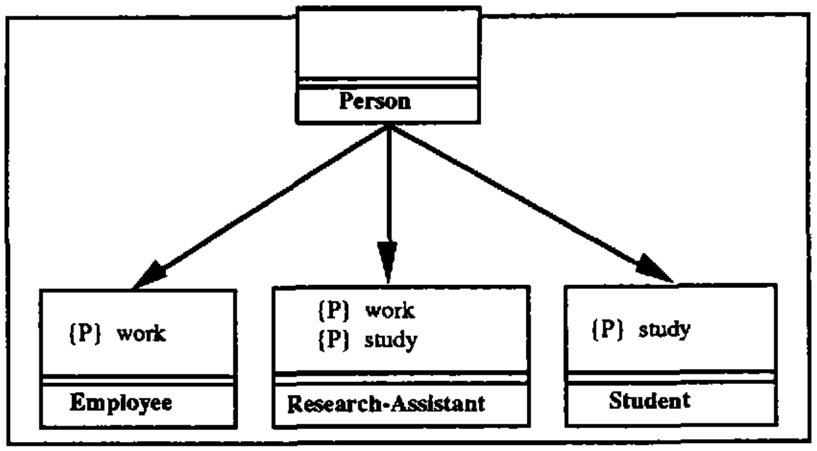


Figure 6-22. Design Class Research-Assistant by Delegation.

Class *Research-Assistant* is no longer a derived class of *Employee* and *Student*, instead, it is also directly derived from the base class - *Person*. Behaviors *work* of *Employee* and *study* of *Student* can therefore be reused in *Research-Assistant* by forwarding. This is presented in Figure 6-23(a), (b), and (c); where (a) shows the type diagram of *Research-Assistant*. The class *Research-Assistant* defines two features; *as-an-employee* and *as-a-student*. Both of these are used for forwarding (see the code example) and are typed as *Person-Pointer*. The type *Person-Pointer*, which is defined in the Predefined Data Type List, denotes a pointer to the type (class) *Person*.

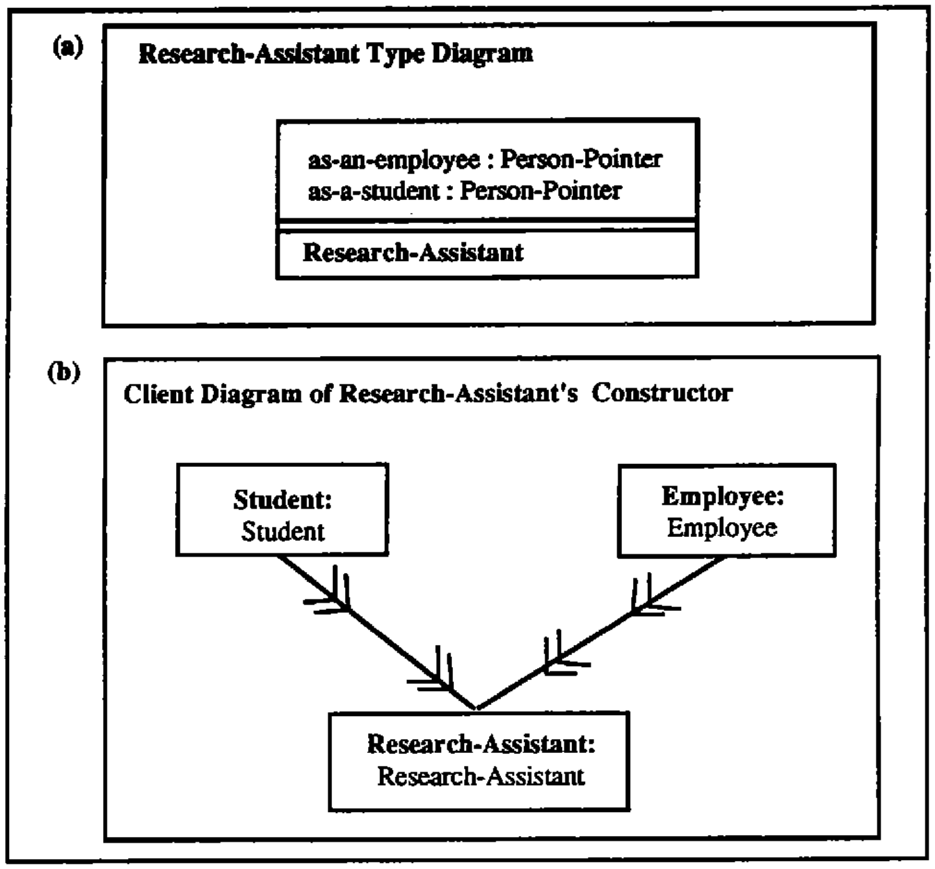

(a)
Research-Assistant Type Diagram
as-an-employee : Person-Pointer
as-a-student : Person-Pointer
Research-Assistant
(b)
Client Diagram of Research-Assistant's Constructor
Student: Student
Employee: Employee
Research-Assistant: Research-Assistant

Figure 6-23. Design Delegation for Class Research-Assistant.

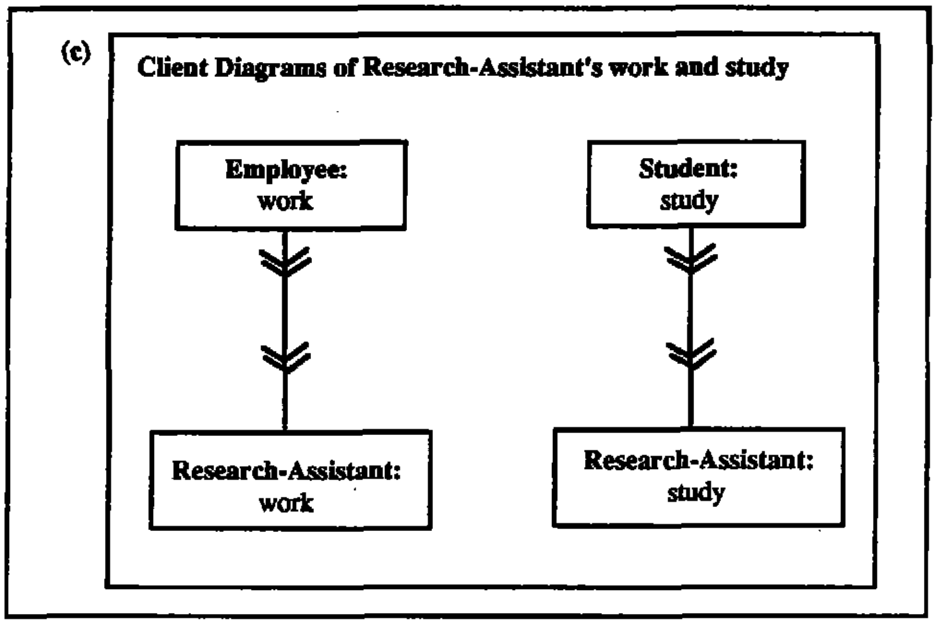

(c)
Client Diagrams of Research-Assistant's work and study
Employee: work
Student: study
Research-Assistant: work
Research-Assistant: study

Figure 6-23. (continued)

Since instances of a derived class are also instances of its base class, we use these two features to store an instance of *Employee* and an instance of *Student* respectively in the *Research-Assistant*'s constructor. Figure 6-23(b) illustrates this; where *Research-Assistant*'s constructor calls *Employee*'s constructor and *Student*'s constructor. Therefore, the feature *as-an-employee* holds a pointer to an instance of class *Employee* and the feature *as-a-student* holds a pointer to an instance of class *Student*. Figure 6-23(c) illustrates the client diagrams for *Research-Assistant*'s *work* and *study*. Both of these are executed by forwarding to the *work* in *Employee* and the *study* in *Student*. The C++ code for this approach is shown as follows:

```
class Person { ..................... };

class Employee : public Person {
public:
 void work (void);
  ...................................
};

class  Student : public Person  {
public:
 void study (void);
   ...................................
};

class  Research-Assistant  {
public:
 Research-Assistant(void)  {                    // as designed in Figure 6-23(b).
  as-an-employee = new Employee;
  as-a-student = new Student;
 };
 void work (void)  {  as-an-employee->work;  };  // as designed in Figure 6-23(c).
 void study (void)  {  as-a-student->study;  };
   ......................................
private:
 Person *as-an-employee, *as-a-student;         // as designed in Figure 6-23(a).
};
```

Reuse by delegation has more run-time flexibility than reuse by inheritance. As shown in the example, we enforce delegation by using forwarding:

```
void work (void)  {  as-an-employee->work;  };
void study (void)  {  as-a-student->study;  };
```

An instance of *Research-Assistant* performs its *work* behavior by forwarding the operation to the behavior *work* of an instance of *Employee* and that instance of *Employee* is pointed by *as-an-employee*. However, unlike inheritance, delegation by reuse works well in simple cases and most importantly, it can be used without violating any abstractions.

6.9.5 Rules of Thumb for Design with Reuse

In summary, the following summarizes the rules of thumb for reuse:

- In general, behavioral features in the classes which are to be put into a class library should be defined as virtual even though they do not have any derived classes in the initial design. This allows the extension of the reuse for these behaviors by inheritance and the design for the dynamic polymorphism (Section 6.5) in the future.

- A good domain analysis is crucial to the reuse intent by discovering the good objects and more importantly, the reusable ones. Reusability often comes from iteratively experimenting with prototypes. Patience and imagination are the keys.

- Cut the class with the right size. "Fat classes" and "skinny classes" are used to serve different reuse intents. "Fat classes" are often reused by public inheritance and "skinny classes" are more likely to be reused by aggregation. Moreover, "fat classes" may be more easily reused in the similar domains, whereas the "skinny classes" may have a broader reuse scope.

- If the intent is reuse of behaviors, one must use abstract base classes. Abstract base classes with virtual functions defined, provide an intuitive and efficient mechanism for behavior reuse. Reuse of behaviors means either adding or rewriting the interface constructed by those virtual functions on a case-by-case basis.

- If the intent is solely for reuse and no subtyping (generalization / specialization) relationship exists between classes, use aggregation instead of inheritance. Aggregation states the reuse intent more clearly than inheritance in this sense.

- If there does exist a subtyping relationship between classes, but the intent is solely for reuse, use private inheritance instead of public inheritance. Public inheritance is used for reflecting class generalization / specialization relationship.

CHAPTER VII

CONCLUSION AND FUTURE EXTENSIONS

In this chapter, we summarize the research conducted for this thesis and present conclusions drawn from the work. We conclude with a discussion of an integrated framework that provides direction for future extensions.

7.1 Conclusion

By discovering and organizing the ontologies of IDEF4 and C++ object models and extending related method concepts, syntax, procedure, and the practice of the method, this research contributes to the construction of a complete implementation design method. The proposed IDEF4/C++ with the addressed techniques is intended to provide an efficient and comprehensive implementation design method for the development of object-oriented software systems in C++.

Three ontological models identified the semantics and terminology of (1) general object model; (2) IDEF4; and (3) C++. These ontologies together specified the boundary of the research domain and defined the primitive concepts and terminology for conducting the research work.

Method concepts were introduced in the discussion of classes, features, and methods, which are identified as the primitive constructs for laying out an IDEF4/C++ design. The notion of classes was introduced through the comparison with the notions of types and objects. The self-referential definition of classes and objects was also clarified. Features - the design constructs used for capturing the characteristics of instances of classes, were introduced in terms of the discussions of feature inheritance, feature presence, feature type, and feature taxonomy. This in turn supported the IDEF4 "least commitment" philosophy. Additionally, the concepts of method sets, contracts, and methods were also introduced.

We extended IDEF4 notations and developed more specific method syntax for IDEF4/C++. A transformation heuristic from IDEF4 to IDEF4/C++ is given, which summarizes the design foci for the transformation. The activities involved in the IDEF4/C++ design procedure are also discussed. In general, each activity employs the processes of partitioning, classifying / specifying, merging / eliminating, and rearranging design artifacts. Moreover, the design steps in each activity are also specified and outlined for conducting an IDEF4/C++ design. We proposed a dynamic model of the configuration management in IDEF4/C++. The development process described in the model starts at an initial design, iterating through the design and implementation processes until the final design and program are released.

We also provided a set of techniques which target the reuse intent in IDEF4/C++. Reusability can be gained through public inheritance (abstract base classes), private inheritance with forwarding, aggregation with forwarding, and delegation. However, if no subtyping relationship between two classes exists, reuse by aggregation is recommended since it states the reuse intent more clear than inheritance.

Without increasing the complexity of the IDEF4 method, IDEF4/C++ takes advantage of C++ language features and best practice experience to bridge the gap between the conceptual design phase and the implementation phase in a software development project.

7.2 Future Extensions

Figure 7-1 in the following page illustrates an integrated framework for object-oriented software system development which provides research directions of the future extensions. The following section discusses thoughts that fabricated this framework and the future extensions that can be derived from the framework:

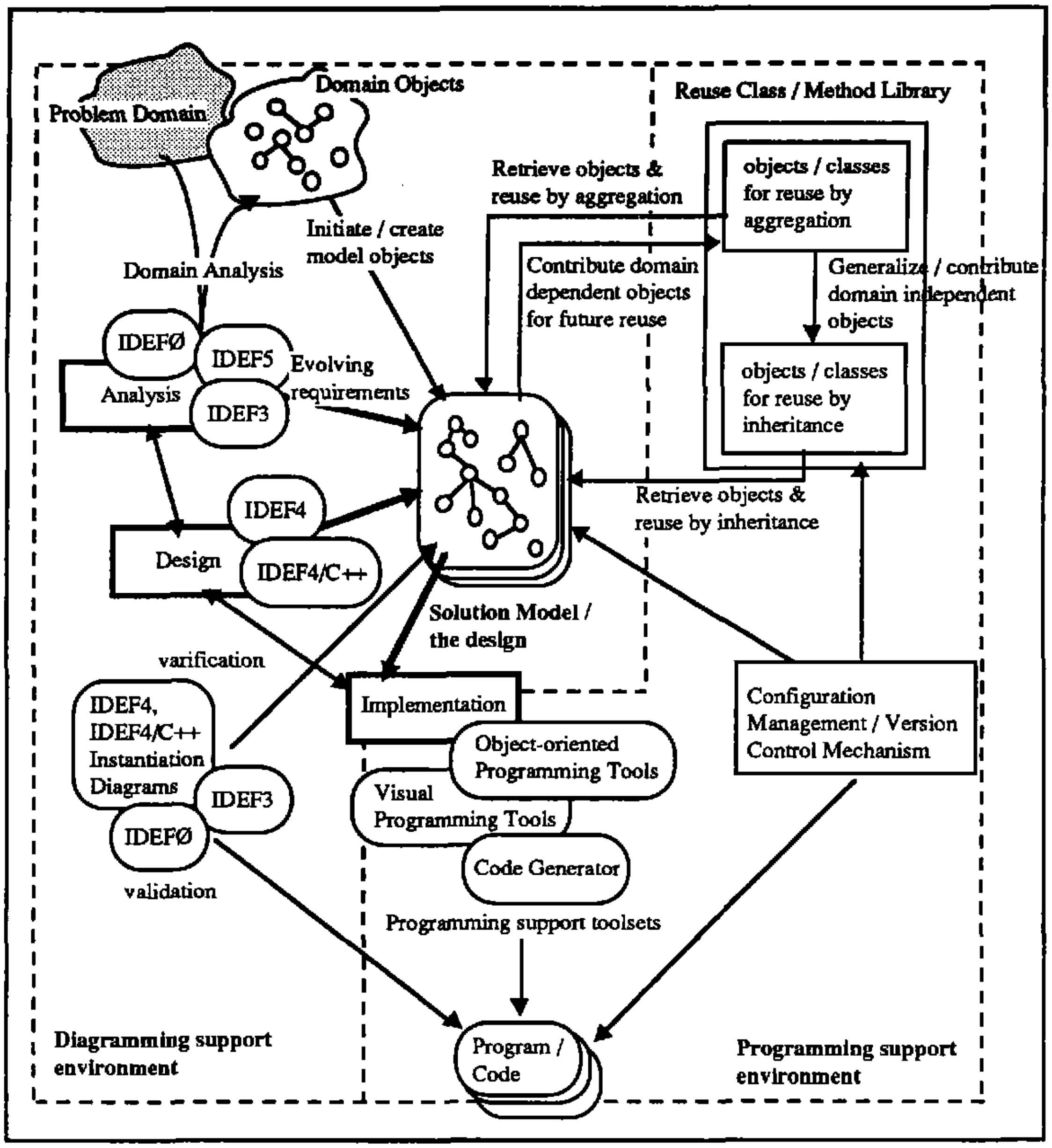

Problem Domain
Domain Objects
Reuse Class / Method Library
Retrieve objects & reuse by aggregation
objects / classes for reuse by aggregation
Initiate / create model objects
Domain Analysis
Contribute domain dependent objects for future reuse
Generalize / contribute domain independent objects
IDEFØ
IDEF5
Evolving requirements
Analysis
IDEF3
objects / classes for reuse by inheritance
IDEF4
Retrieve objects & reuse by inheritance
Design
IDEF4/C++
Solution Model / the design
varification
Implementation
Configuration Management / Version Control Mechanism
IDEF4, IDEF4/C++ Instantiation Diagrams
Object-oriented Programming Tools
IDEF3
Visual Programming Tools
IDEFØ
Code Generator
validation
Programming support toolsets
Diagramming support environment
Program / Code
Programming support environment

Figure 7-1. An Integrated Framework of Object-oriented System Development.

- The concept of Computer Aided Software Engineering (CASE) has occasionally been associated with two different development cultures: *Environments* culture (programming support environment) and *Diagramming Tools* culture (Schefström 93). The former culture, also called "Back-End CASE", emphasizes the later stages in the software life cycle (implementation, test & verification). Whereas the later culture, such as IDEF4 and IDEF4/C++, has more of an industrial and administrative flavor and emphasizes the earlier software development stages (analysis, design, and implementation). This, of course, is based on the philosophy that a good design makes the coding trivial. However, evidence has shown that the difference between these two cultures decreases as both cultures are gradually integrated (Schefström 93). The CASE tools have widened their scope, often by attempting integration with code generation or programming support toolsets (i.e., reusable class libraries). We follow this understanding, and construct the integrated framework.

- An object model (IDEF4 or IDEF4/C++) only provides a static object structure viewpoint towards the solution design. However, to accomplish a complete design of the solution model, other perspectives, such as the system's dynamic behavior (IDEF3) and functional decomposition / architecture (IDEFØ), also have to be provided. One of the future research directions, therefore, is to provide a platform for the integration of these IDEF methods. The platform should provide an automate transforming mechanism between the artifacts specified in each perspective (model).

- The primitive software constructs for an object-oriented system are classes and methods. These constructs are the nature reuse modules. Generally, methods are reused in terms of incorporating their defining classes into the design. An efficient class library management system is therefore crucial. Such a system is more than a class browser. The ability for controlling changes and the support for the evolution of reusable constructs should be provided. A dynamic model, which includes the reuse concerns into the development process, should be developed for constructing such a system.

- As shown in Figure 7-1, a complete configuration management system is more than a source code version control device in the integrated framework. The control levels should be multiple and flexible. The configuration item (as defined in Section 5.4) can be the system, its subsystems, components, a class, or even a method.

- To integrate two different CASE cultures, one effort is to construct an efficient and seamless transforming mechanism between the method support environment and those programming support toolsets. The mechanism extends the method culture and is especially beneficial to an implementation design method (such as IDEF4/C++) to attain code generation ability. Language supports should be developed as modules in the mechanism to gain extensibility and malleability for the integrated framework.

REFERENCES

Atkins, M.C., and Brown, A.W., 1991, Principles of Object-oriented Systems, In McDermid, J.A., editor, *Software Engineer's Reference Book* (Oxford, England: Butterworth-Heinemann Ltd), Chapter 39.

Boehm, B.W., 1976, Software Engineering. *IEEE Transactions on Computers*, **C-25**, 1226-1241.

Boehm, B.W., 1988, A Spiral Model of Software Development and Enhancement. *IEEE Computer*, **21**, 61-72.

Booch, G., 1991, *Object Oriented Design with Applications* (Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc.).

Coad, P., and Yourdon E., 1991, *Object-Oriented Design* (Englewood Cliffs, NJ: Yourdon Press).

Coplien, J.O., 1992, *Advanced C++ Programming Styles and Idioms* (Reading MA: Addison-Wesley Publishing Co.).

DeMichiel, L.G., 1993, CLOS and C++: A Comparison, In Paepcke, A., editor, *Object-Oriented Programming: The CLOS Perspective* (Cambridge, MA: The MIT Press), pp. 157-180.

Henderson-Sellers, B., and Edwards, J.M., 1990, The Object-oriented Systems Life Cycle. *Communications of the ACM*, **33**, 142-159.

Kim, W., Bertino, E., and Garza, J.F., 1988, MCC Technical Report: Composite Objects Revisited (Austin, TX: Microelectronics and Computer Technology Co.), ACA-ST-387-88, pp. 15-19.

Korson, T., and McGregor, J.D., 1990, Understanding Object-Oriented: A Unifying Paradigm. *Communications of the ACM*, **33**, 40-60.

Lippman, S.B., 1991, *C++ Primer* (Reading, MA: Addison-Wesley Publishing Co.).

Mayer, R.J., and Edwards D.D., 1990, IDEF4 Technical Report (College Station, TX: Knowledge Based Systems Laboratory).

Mayer, R.J., Keen, A., and Wells, M.S., 1992a, IDEF4 Object Oriented Design Method Report (Dayton, OH: Integrated Information Systems Evolution Environment Project, United States Air Force AL/HRGA, Wright-Patterson Air Force Base), AL-TR-1992-0056.

Mayer, R.J., Menzel, C.P., and Mayer, P.S.D., 1992b, IDEF3 Process Description Capture Method Report (Dayton, OH: Integrated Information Systems Evolution Environment Project, United States Air Force AL/HRGA, Wright-Patterson Air Force Base), AL-TR-1992-0057.

Mayer, R.J., Benjamin, P.C., Fillion, F., Futrell, M.F., and deWitte, P.S., 1992c, IDEF5 Method Report (Draft) (Dayton, OH: Integrated Information Systems Evolution Environment Project, United States Air Force AL/HRGA, Wright-Patterson Air Force Base).

Menzel, C.P., and Mayer, R.J., 1991, IDEF5 Concept Report, Final Technical Report (Dayton, OH: Integrated Information Systems Evolution Environment Project, United States Air Force AL/HRGA, Wright-Patterson Air Force Base).

Meyer, B., 1987, Reusability: The Case for Object-Oriented Design. *IEEE Software*, March, 50-64.

Meyer, B., 1988, *Object-Oriented Software Construction* (Englewood Cliffs, NJ: Prentice Hall).

Nelson, M.L., 1990, An Object-oriented Tower of Babel (Monterey, CA: Naval Postgraduate School), Technical Report.

Object Management Group, 1991, The Common Object Request Broker: Architecture and Specification. OMG Document Number 91.12.1 Revision 1.1.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., 1991, *Object-oriented Modeling and Design* (Englewood Cliffs, NJ: Prentice Hall).

Schefström, D., and van den Broek, G., Editor, 1993, *Tool Integration: Environments and Frameworks* (Chichester, England: John Wiley & Sons Ltd.).

Snyder, A., 1986, Encapsulation and Inheritance in Object-Oriented Programming Languages. *OOPSLA '86 Proceedings*, September, pp. 38-45.

Snyder, A., 1990, An Abstract Model for Objec-Oriented Systems (Palo Alto, CA: Software Technology Laboratory, Hewlett-Packard Laboratories), Report STL-90-22.

Snyder, A., 1993, The Essence of Objects: Concepts and Terms. *IEEE Software*, **10**, 31-42.

Stroustrup, B., 1990, *The Annotated C++ Reference Manual* (Reading, MA: Addison-Wesley Publishing Co.).

Wirfs-Brock, R.J., and Johnson R.E., 1990, Survey on Current Object-oriented Design. *Communications of the ACM*, **33**, 104-124.

APPENDIX A

Project: Ontology of Object-oriented technology Version: 2	Analyst: Howard Date: 6 / 30 /93	Term Glossary	Reviewer: Date:	Document Number:

Term #	Term	Glossary
	Abstract class	A class that is created only for the purpose of inheritance or for defining methods and attributes that will be inherited by lower-level classes.
	Abstraction	The process of only focusing on the essential characteristics of an object that distinguish it from other objects in a specified domain.
	Actor object	An object that can send message to other objects. It is a synonym of "sender object".
	Agent object	An agent object sends messages to other objects and receives messages from other objects as well.
	Association	Association is a relationship between two or more classes describing the semantics hold by them.
	Attribute	An attribute is a data variable held by the objects in a class. Each attribute has a value for each object instance. In IDEF4, the term "attribute" has different meaning. An attribute is a value-returning feature, and it can be further categorized into a slot or a function.
	Base class	In C++, a "base class" refers to a superclass.
	Behavior	Object behavior specifies how an object acts and reacts, and how the state changes in terms of message-passing.
	Class	A template for defining methods and attributes for a particular type of objects. All objects of a given class are identical in data structure and behavior but contain different values for their attributes.
	Class hierarchy	A tree structure representing the inheritance relationship among a set of classes.

Term #	Term	Glossary
	Class operation	An operation can be operated on and by the class itself. Methods like constructors can only be applied by and on the class since the objects being operated on haven't been created yet. Examples such as a query for the summary information of the class (how many instances in this class?), or a browsing function for a list of attributes and methods of a class.
	Class variable	A class variable is an attribute especially used to describe the class structure. It is shared by all the instances of the class. It is implemented in C++ in terms of "static member" declaration.
	Composite object	An object that contains one or more other objects, typically by storing references to the objects as the return values of its features.
	Constructor	A constructor is a method that creates instances (objects) of the class and/or initializes their states (by giving attributes values). Constructors use the class name as the function name. In C++, constructors can be overloaded.
	Data member	The implementation of the attributes defined in a C++ class definition.
	Derived class	In C++, a "derived class" refers to a subclass.
	Destructor	A destructor is a method that deletes objects and free the memory they use. Destructors can be overloaded in C++. Destructors use the class name as the function name.
	Dynamic polymorphism	The invocation of a method is not determined until the run time. Example such as C++ function overriding.
	Encapsulation	A mechanism in which data (attributes) is packaged together with its corresponding procedures (methods). In object-oriented technology, the mechanism for encapsulation is the object.
	Feature	In IDEF4, the term feature is a generic term represents either an attribute or a method.

Term #	Term	Glossary
	Friend	Function or class that is declared as friend to a class can access private definition of the class. This mechanism is defined in C++ to give a flexibility on sometimes over-restricted information-hiding rules.
	Generic Function	The term "generic function" is a synonym used in CLOS to refer to "message". In CLOS, user can define a "message" (a generic function) by using the function "defgeneric". Note that the methods that can be invoked by a generic function use the same name as the generic function.
	Information hiding	The technique of making the internal details of a module inaccessible to other modules, protecting the module from outside interference, and preventing other modules from relying on details that might change over time.
	Inheritance	A mechanism whereby classes can make use of the methods and attributes defined in all classes which are their ancestors in the structure of the class hierarchy. Inheritance refers to the mechanism of sharing attributes and methods using the generalization relationship. In C++, "inheritance" is also referred to as "derivation".
	Instance	A term used to refer to an object that belongs to a particular class.
	Instantiation	Instantiation is the process that creates instances from a class (metaclass as well).
	Iterator	An operation that controls iteration over a range of values or a collection of objects. For example, sort operation of a queue.
	Link	A link is an instance of an association. It is a "physical or conceptual connection between objects" (Rumbaugh).
	Member function	The implementation of a method defined in a class is referred to as a member function in C++.

Term #	Term	Glossary
	Message	A signal from one object to another that requests the receiving object to carry out one of its methods. A message consists of three parts: the name of the receiver object, the method it is to carry out, and the parameters the method may require.
	Metaclass	Metaclass is a class for describing the structure and behavior of other classes. Its instances are themselves classes.
	Metaobject	Instances of metaclasses are themselves classes, but they can also be considered as objects. These classes are called metaobjects. Metaobjects contain class attributes and class operations (methods) that can help to manipulate and query the structure and behavior of the class that is intended to describe.
	Method	A procedure attached to an object that is made available to other objects for the purpose of requesting services of the owner object. Most communication between objects takes place through invoking methods.
	Multiple inheritance	A scheme for structuring inheritance relationship among classes where each class can have any number of superclasses.
	Multiple polymorphism	The invocation of a method is based on more than one parameters. Examples such as C++ function overloading or CLOS multi-methods.
	Object	A software packet containing a collection of related attributes (variables) and methods (functions / procedures). The term is used inconsistently in the literature, sometimes referring to instances and other times to classes. The term object refers to a specific instance of a class and possesses the characteristics of that class.
	Operation	An operation simply refers to a request (message) that may be applied to or by objects in a class. It is a synonym of "message".

Term #	Term	Glossary
	Overloading	The assignment of multiple meanings to the same method, allowing a single message to invoke different methods depending on the number and types of parameters accompanying it.
	Overriding	A special case of overloading in which the same name is given to a method or variable at two or more levels on the same branch of a class hierarchy. The name of the method which is the lowest in the hierarchy takes precedence, overriding the more general definitions (methods) further up in he hierarchy.
	Parameterized class	Parameterized class provides a template for creating other classes. Similar classes (array of integer, array of string) can be created from same template by filling in different values for the parameters that the template carries. The term "generic class" is a synonym of "parameterized class".
	Polymorphism	The mechanism to hide different implementations behind a common interface, simplifying the communications among objects. Polymorphism means that the same operation may behave differently on different classes (objects).
	Private	A declaration specifies that the declared features are accessed only by their owner class. Note that IDEF4 has a different scope for general "private" definition. In IDEF4, private features can be accessed by their owner class and also all the derived classes.
	Private derivation	In a private derivation, the inherited nonprivate features of the base class become private features of the derived class.
	Property	Properties of an object is a synonym of "attributes". Both are defined for associating values.
	Protected	A declaration that lets the declared features can be accessed only by their owner class and the direct subclasses. (C++)

Term #	Term	Glossary
	Protected derivation	A protected derivation lets the inherited nonprivate features from the base class become protected features of the derived class. (C++)
	Public	A declaration specifies that the declared features are accessed by every class in the program.
	Public derivation	In a public derivation, the inherited nonprivate features of the base class become public features of the derived class.
	Return value	An object or a data type that a receiver object passes to a sender object to respond to that message.
	Routine	In IDEF4, "routine" is used to refer to a feature which is computational-initiating. The term "routine" is a synonym of "method", which is used to implement object behavior. In the late design phase, a routine can be further specified as a function or a procedure.
	Single inheritance	A scheme for structuring inheritance relationship among classes so that each class has only one superclass. Single inheritance assures that all class hierarchies will conform to a simple tree structure.
	Single polymorphism	The invocation of a method is based only on the name of the receiver object. Example such as C++ function overriding.
	Static polymorphism	The invocation of a method is determined at compile-time. Example such as C++ function overloading.
	Virtual function	In C++, only the class member functions can be declared as virtual. Virtual functions, which are bound dynamically at run-time, provide a way of hiding (encapsulating) the implementation details of a class inheritance hierarchy from programs that make use of the class hierarchy. Note that, only the member functions that are declared as virtual can be overridden by subclass.

VITA

LI-TSUNG HSIEH received a B.S. degree in Industrial Engineering from Tunghai University, Taichung, Taiwan in 1988. He was a consulting assistant for projects entrusted by the Ministry of Economic Affairs in Taiwan from 1987 to 1988. He served in the Army from 1988 to 1990. From 1990 to 1991, he worked as a full-time teaching assistant with the Computer Center of the College of Management at Tunghai University. He is currently a research assistant in the Knowledge Based Systems Laboratory of the Industrial Engineering Department at Texas A&M University. His research interests include object-oriented system development, modeling methodology, and expert system applications. His permanent address is 4F NO 3 LN 3, Hsintung Street, Taipei, Taiwan.