Dee: an Object Oriented Programming Environment and its Implementation

Benjamin Yik Chi Cheung

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfilment of the Requirements for
the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

March 1992

ISBN 0-315-81001-7

Abstract

Dee: an Object Oriented Programming Environment and its Implementation

Benjamin Yik Chi Cheung

We have combined a strongly-typed object oriented language with an integrated, interactive development environment. In this environment, Dee, we designed the compiler as an integral component of the environment. Coupling the compiler and the browser simplifies symbol table management in the compiler. Conversely, the same coupling ensures that information is semantically checked before the browser displays it. Also, programmers do not have to understand the class hierarchy because the compiler creates class views.

In Dee, the class interface manager is the fundamental key idea of the whole system. It provides different *views* of class interface to other system components. It facilitated both the development of the compiler and the environment.

In order to support dynamic binding, most of the object oriented languages payed a high cost on memory space and CPU time for message dispatching. In Dee, an improved color index technique was implemented. It shows that we can use heuristics instead of exhaustive searching to get better space consumption.

Acknowledgments

I would like to express my sincere gratitude to Dr. Peter Grogono my thesis supervisor, for his guidance and valuable insight throughout this research. His support and patience were invaluable in the preparation of this thesis.

I thank Dr. Peter Grogono and Dr.Hafedh Mili for their financial support.

I thank all people in the Dee group: Lawrence Hegarty, Joseph Yau and WaiMing Wong for stimulating discussion, and make our project goes smoothly, and led to an improvement of this thesis. All my friends, D. Pao, C.H. Yim, S.Y. Leung, K.L. Ma, S.C. Leung, K.O Lau and A.Wongyai have helped brighten my days at Concordia.

Specially, I would like to thank my girlfriend Min Huang, for her patient, understanding and encouragement.

Finally, I am most grateful to my parents, for their encouragement, support and love.

This research was supported in part by a grant from the Natural Sciences and Engineering and Research Council of Canada

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The need to develop, reuse and maintain large complex software systems in a competitive and dynamic environment has created interest in a new approach, the object oriented paradigm, for software design and development.

Problems with traditional development using the classical life cycle include few facilities for iteration, no emphasis on reuse, and no unifying model to integrate the development phases[KM90]. There is little opportunity for code reuse and maintenance costs account for a very large share of total system costs.

The object-oriented paradigm was proposed to address the reusability and maintaintablity of software system and development.

1.1 The Object Oriented Programming

The goal in designing individual software components is to describe a concept in such a way that it can eventually be an executable form. The Abstract Data Type is the object based paradigm technique for capturing this *conceptual information*. The *class* is the conceptual modeling tool we are using:

- A computation is performed by a set of objects.

- Objects communicate by sending **messages** to each other.

- An object will execute a **method**, if it has received a message.

Objects, Classes, Instances

The object oriented paradigm: an *object* combines data and a set of operations (methods)[SB86, Bud91, KL89]. It stores the data in instance variables in order to save its state, and responds to carry out a specified procedure according to a received message; a *class* is the static representation of the object, it specifies a set of visible operations, a set of hidden *instance variables* and hidden *procedures* which implement the operations. All the operations of a class are shared by objects which are called **instances** of the class.

When a new instance of a class is created, it has its own set of instance variables, but shares all the operations with other objects of that class. The instance variables of the object can only be modified indirectly by invoking the operations.

Methods, Messages, Receiver

A **method** is the function that implements the response when a message is accepted by an object.

A **message** is the specification of an operation to be performed on an object. Similar to a procedure call, the operation to be performed is named indirectly through a *selector* whose interptertation is determined by the class of the object.

A **receiver** is the object which will receive the messages.

Basically, we can use the following notations to describe the relationship between method, message and receiver. For any given message, M, we write $r.m(a, b, ..)$: where

r is the *receiver*, m is the name of the *method* and $a, b, \ldots$ are arguments.

Inheritance, Typing

Inheritance is strictly a *reusability* mechanism for sharing behavior between objects. For example, if class B inherits class A, then class B will have all or most of the behavior and properties of A. There are a lot of forms of inheritance depending on the implementation and the object modeling concept of particular language. Basically, inheritance involves the following issues[Nie89]:

- What properties can be inherited?
- Is multiple inheritance supported?
- Does inheritance occur statically or dynamically?
- Which inherited properties are visible to the client?
- Can inherited properties be overridden or suppressed?
- How are conflicts resolved?

In an object oriented programming language, an object *type* is superficially the same thing as an object class. In general, programming languages are commonly classified according to the extent of type checking provided at compile time. Object oriented languages vary widely on this issue. We have typeless object oriented language like SmallTalk[Gol84], CLOS[1][Kee89]; and strongly typed language like Eiffel, C++[Str86] and Dee.

[1]CLOS is short for Common Lisp Object System.

For strongly typed object oriented language, type checking is applied at the level of an object. Objects not only have data, but are also associated with operations. In the object oriented world, type checking must not only be concerned with the interpretation of data, but also needs to determine which operations may be applied to an object.

Informally, one type, B, *conforms* to another, A, if some subset of its *interface* is identical to the other. We then say that B is a *subtype* of A. Therefore, the relationship between A and B may be *equivalent* or *included.*

Obviously, the type confirmation mechanism depends on the components of the interface of an object class. And the interface depends on the particular type model chosen for a language.

In general, suppose that class B inherits from class A. We say that B is a subtype of A. The compiler will allow an instance of B to be assigned to an instance of A, in the form of:

$$a := b \tag{1.1}$$

where a and b are instances of A and B.

Polymorphism and Dynamic Binding

A polymorphic function is one that can be applied uniformly to a variety of objects. For example, the + function, may be used to add two integers, two strings, or two floats. Polymorphism in object oriented languages is concerned with a set of operations that coincidentally share a name, but have completely different behavior. This is *mere* overloading of operation names.

Inheritance is close to polymorphism, because the same operations that apply to instances of a parent class also apply to instance of its subclasses (children).

Polymorphism enhances software reusability by making it possible to implement generic software that will work not only for a range of existing objects, but also for later modification or objects to be added later. For examples, A *Sorting* class will sort any list of objects that support comparison operations such as *equal*, and *less_than*.

In object oriented programming languages, if variables can be dynamically bound to instances of different object classes, we call it *dynamic binding*. Since we can not determine the methods to be executed at compile-time, some forms of *run-time method lookup* or *message dispatching mechanism* must be performed.

For typeless language like Smalltalk, the method lookup mechanism makes it necessary to search through the class hierarchy at run-time to find the method of an inherited operation. For strongly type languages like Eiffel, C++ and Dee, a run-time support system will be created by the linker according to the classes needed at the run-time. The invocation of functions on instances will be resolved at run-time depending on the class to which the instance belongs. The resolution mechanism is called *message dispatching mechanism*[2].

1.2 Languages and Environments

1.2.1 Simula

Simula[BDMN73, ND81] was designed in 1967, under the name Simula 67, by Ole-Johan Dahl and Krysten Nygaard. The name was shortened to Simula in 1986. Recent converts to the ideas of object oriented programming sometimes think of Simula as a respectable but defunct ancestor.

[2]See the Chapter on The Linker in section 4.3

Simula is an object oriented extension of Algol 60. Its basic control structures and data types are drawn from Algol 60.

Simula made several contributions to the object oriented paradigm. For example, it supported:

- *Single inheritance*: B is declared to be an heir of A by

  A Class B; begin ... end

  Here A is parent, B is the sub-class;

- *Information hiding*: a feature declared as *protected* will not be available to the clients and a feature declared as *hidden* will not be available to proper descendants.

- *Virtual routine* is the deferred mechanism provided by Simula. A routine can be defined as a *virtual* routine in a parent class and implemented in the descendant classes.

- *Polymorphism* is supported. By default, binding is static rather than dynamic, expect for virtual routines.

The implementation of Simula supports garbage collection. The language definition does not include a standard class library.

1.2.2 Smalltalk

The ideas for Smalltalk[GR83] were formulated around 1970 by Alan Kay. Adele Goldberg, Daniel Ingalls are two other persons who made key contributions to its implementation at Xerox.

As a language, Smalltalk combines the influence of Simula with the free, typeless style of Lisp. It emphasis is on dynamic binding. No type checking is performed.

Everything in the Smalltalk system is an object; a class is an object and is viewed as an instance of a higher-level class called a *metaclass*. By this approach, Smalltalk obtains some benefits. They include:

- conceptual consistency, a simple object abstraction is applicable to both classes and objects.

- contributions to the programming environment. At run-time, classes are part of the data. This facilitates the development of compiler, debugger, browser and the source inspector. This is appropriate in an interpreted environment.

- Considering a class as an instance variable of a metaclass makes the run-time system easier to define or modify an individual class rather than its instances and makes it possible to define a new method during run-time.

Smalltalk is not just a language but a graphical, interactive programming environment. It popularized many of the advances in this area such as multiple windows, icons, integration of text and graphics, pull-down menus and the use of a mouse as a selecting and pointing device[Gol84]. A very important window based tool, the *browser*, is incorporated in this environment. It allows users to retrieve and view class or system information during the development period; and it also allows users to inspect a specified instance variable.

This environment is Smalltalk's most significant contribution.

1.2.3 Lisp: CLOS And Flavor

Lisp[Fat88, McC81] is a functional language widely used in the artificial intelligence community. By itself Lisp has a unique computing concept. Both data and program have a uniform representation as lists. On the other hand, most of the implementation of Lisp language already has incorporated the concepts of advanced programming environments such as garbage collection, tree like-data structures, editing tools, debugging and tracing facilities. Thus it is not surprising that Lisp has been used as the base language for several object oriented languages. A good Lisp environment is directly applicable to the implementation of object oriented concepts.

There are two popular object oriented extensions of Lisp. They are COMMON LISP OBJECT SYSTEM (CLOS)[KEE89] and FLAVORS LISP. CLOS has now become a standard part of the COMMON Lisp. The Flavor[INC88, INC91] object system is developed by Franz Inc, it almost have identical functionalities and function protocols to CLOS.

CLOS

CLOS comprises a set of tools for developing object oriented programs in Common Lisp. It has been adopted as part of Common Lisp by the X3J13 committee[3]. The major techniques supported by CLOS are listed below:

Classes and Instances: CLOS supports multiple inheritance. Each class is a Lisp type in the language with two object oriented properties, data slots and methods. Specifically, it allows instance variables to have a default value, after their creation.

[3]X3J13 is a committee working on creating the ANSI Standard Common Lisp

Methods: CLOS allows users to define methods with different roles. Besides the *primary methods*, users can define *before methods*, *after methods* and *around methods*. These provide an extremely flexible framework that enables the users to define code in modules and reduce code repetition fragments in a large software system.

Dispatching mechanism: In CLOS, the system provides generic message dispatching rules. Users can redefine the dispatching sequence and control the combination of methods in the inheritance hierarchy. These facilities give users freedom to control how the methods are called and what is the update sequence of the updating of an objects state or value.

CLOS allows redefinition of generic functions, methods and classes on the fly. CLOS ensures that everything that is affected by the redefinition is automatically updated.

Due to the goal of designing CLOS, which is to *satisfy most application*, according to its specification and implementation, all CLOS features add up to a great deal of expressive power, and the programming interface is very flexible. The disadvantage of this flexibility is that the CLOS user might be overwhelmed by the wide assortment of techniques and features to be learned. In particular, CLOS often supports more than one way of doing a single thing, and there is not always a clear guideline as to which way is preferable.

1.2.4 Eiffel

Eiffel[Mey90a, Mey88] was designed by Bertrand Meyer of Interactive Software Engineering. It is a strongly typed object oriented language.

For an object oriented language, Eiffel was strongly influence by Simula, not only the conceptual model but also the syntax and style of the language. Meyer invented Eiffel based on the object oriented design. As an object oriented language, Eiffel supports multiple inheritance, deferred methods, generic classes, renaming and redefinition.

The most novel ideas in Eiffel are the concepts of assertions -- *programming by contract.*

An assertion is a property of a programs entity. For example, an assertion may express that a certain integer has a positive value or that a certain reference must be void.

Syntactically, assertion are simple boolean expressions, with a few extensions. For example:

$$n \geq 0; \text{not x.void}$$

The semicolon is equivalent to a logical *and.*

Eiffel uses assertions for the semantic specification of methods or classes. Consequently, a method in Eiffel is not only a piece of code which performs a specified task, but it may also have two assertions associated with it: a *precondition* and a *postcondition.* The precondition expresses the properties that must hold whenever the method is called; the postcondition describes the properties that the method guarantees when it returns.

In Eiffel, users also can define an assertion for a individual class, called a *class invariant.* A class invariant is a global property of all instances of a class, which must be preserved by all its methods. An invariant for a class C is a set of assertions that is satisfied by every instance of C at all *stable* times. Meyer defines that the stable time of a object instance should be: on instance creation and before or after every method execution. A simple example is the following property of class STACK2 which in the standard class library of Eiffel:

$$0 \leq nb\_elements; nb\_elements \leq max\_size$$

The concept of assertion enhances the softwares reliability and it can pay a crucial role in helping programmers write correct programs. In particular, it helps the class designer to make the specification more clearly. At the same time assertions give programmers a more accurate way to understand existing systems.

1.2.5 C++

C++[Str86, ES90, Sak88] was designed by Bjarne Stroustrup of AT&T. C++ is a hybrid language which combines the feature of imperative and object oriented language. The language design is an attempt at *a better C*. Programmers need not do all their coding in the object oriented way. It allows programmers to code routines in C. Almost any correct C program is a correct C++ program.

C++ supports only single inheritance in its published descriptions. Dynamic binding is also available, but only for those routines that have explicitly been marked as *virtual* in their original class. This makes it easier for the compiler to implement an efficient message dispatching mechanism. But it loses the flexibility of redefinition and extendibility of existing classes.

The weakest point of C++ is, that it does not support any memory management facility. No garbage collector is provided; programmers are responsible for managing the object memory by writing constructor and destructor functions.

1.3 Dee

Two of the major problems of software development are code management, software extendability and code reuse. Although object oriented programming has been widely acclaimed as a way of solving these problems, the potential benefits of the object oriented paradigm have not yet been realized in currently available products.

The typeless object oriented languages, such as Smalltalk and CLOS, have a nice programming environment, but they lack run-time safety and are too slow for general applications. They are suitable for prototyping environment. On the other hand, most of the strongly typed object oriented languages, like Eiffel and C++, do not have a good programming environment.

The design goal of Dee is: to design a simple yet complete object oriented programming language with an interactive programming environment which satisfies the requirements of object oriented development. The significant differences in the design approach between Dee and other languages are: we design our language and programming environment at the same time and implemented them in parallel. In most previous case, the language was designed and implemented first, followed by extra tools for their programming environment.

The primary goals of designing and implementing Dee were:

1. a programming language which provides full support for the object oriented paradigm;

2. a programming environment for the language which supports all phases of software development; and

3. a library of classes which facilitate both coding at a high level of abstraction

and efficient object code.

The detail issues of designing Dee can be found in [Gro91b]

The orginal version of Dee was designed by Dr Peter Grogono. He implemented the first prototype system PC-Dee in Turbo Pascal. The second revision of PC-Dee incorporated the concept of *class interface manager.* It was also written in Turbo Pascal and the class interface manager was implemented by Benjamin Cheung during the summer of 1990. A set of standard class libraries were also developed in this version in order to get real development experience.

After evaluating the PC-Dee[Gro90], we redesigned and implemented Unix-Dee[4]. For the language itself, we made some small changes which are the result of the responses of student users to PC-Dee. We tried to climate mistakes in PC-Dee and enhance the integration and performance of the whole system.

The goal of first phase development are:

- Develop a new compiler for Dee on Unix environment;
- Redesign the Class Interface Manager;
- Build an interactive development programming environment, including a customized editor, a semantic browser and a graphical user interface;
- Redesign the object linker to get better space and run time performance.

1.3.1 System architecture

Figure 1.1 shows the major functional components of our system. They include: a compiler, consisting of a scanner/parser, a semantic analyzer, and a code generator.

[4]In the following chapters of this thesis, Dee denotes Unix-Dee

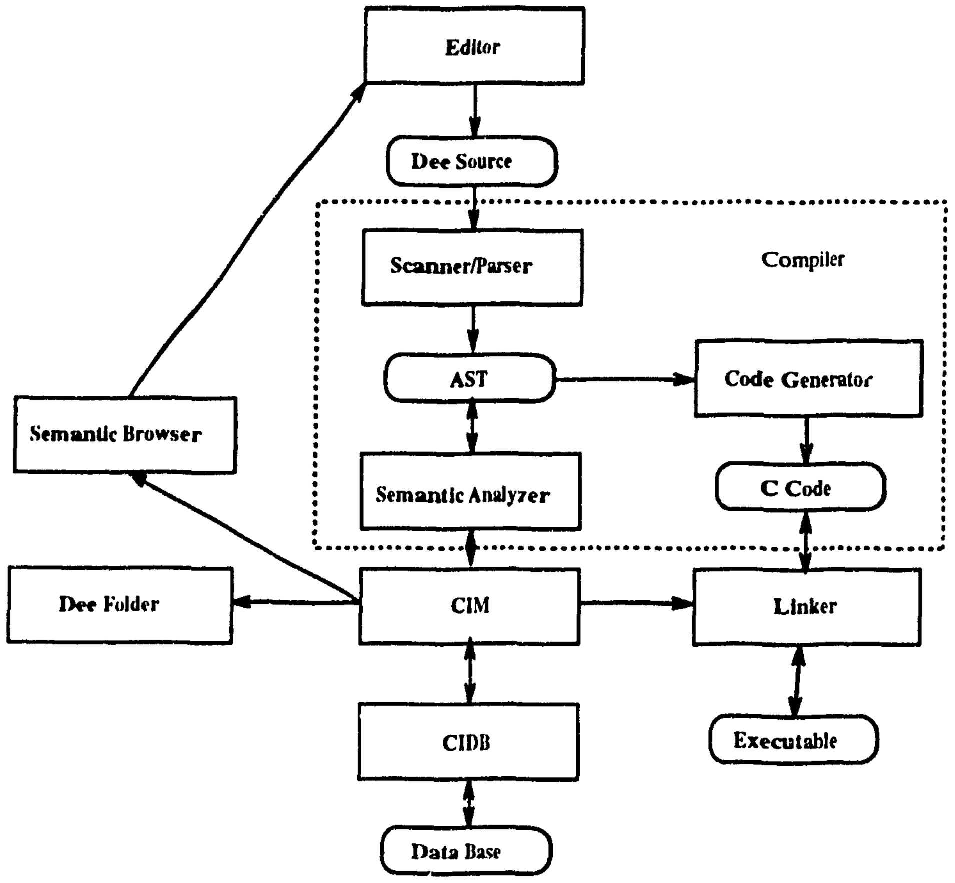


Figure 1.1: Dee System Organization

This compiler translate the Dee code to C; the class interface manager consists of data base maintenance unit and class interface database; and the linker, which is an object linker which creates the run-time system of each individual executable program. The diagram shows that the whole system is tightly coupled and highly integrated. This is our key design approach, each component should be integrated and benefit from the others.

The Compiler

The scanner/parser is responsible for translating the Dee source text into an intermediate data structure, called the abstract syntax tree, and denoted by AST in Figure 1.1. The semantic analyser is responsible for semantic checking and enriching the information of the original AST by retrieve related information from the Class Interface Manager (CIM). After semantic checking, the semantic analyser saves the class interface to the CIM; The code generator translates the Dee into C;

The Class Interface Manager[5]

The CIM, is a functional unit which supports interface management, system configuration and database mainteinance facilities to the environment. It is responsible for storing class interface into the class interface data base (CIDB), provide service interface, *views*, to other components.

The class interface data base [6] is a specially designed database module, which provides database service to the CIM.

The Semantic Browser

The semantic broser[CG92] is a tool which can be using to retrieve class interface and semantic information from the CIM. It can be activated in a text terminal or inside the editor.

Dee Folder

The Dee folder is a graphical user interface which supports class hierarchy browsing, semantic and source browsing.

The Linker

The linker of Dee is the functional unit which creates the run-time data structure and support environment for each Dee program. For an individual program, the linker will try to determine: the smallest set of classes which need to

[5]Class Interface Manager will be abbreviated to CIM from now on
[6]Class Interface Data Base will be abbreviated to CIDB from now on

Modules	Person
Emacs Dee Mode (editor)	Lawrence Hegarty
Scanner/Parser	Lawrence Hegarty
Semantic Analsyer	Waiming Wong, Lawrence Hegarty
Code generator	Lawrence Hegarty
Garbage collector	Lawrence Hegarty
Class interface manager	Benjamin Cheung, Joseph Yau
Linker	Benjamin Cheung
Semantic Browser	Benjamin Cheung
Dee Folder (GUI)	Benjamin Cheung
Standard Class	Peter Grogono, Lawrence Hegarty, Benjamin Cheung

Table 1.1: **Project: Module and People**

be linked together; and the best space consumptic . The linker will translate all run-time supporting system into C data structure in a C source file, then produces a Unix *make* file which is responsible for using the C compiler and Unix-linker to create the executable program of Dee.

1.3.2 The project

Table 1.1 shows, that the Dee project had been divided into several major components and it was implemented by four graduate students. They are Benjamin Cheung, Lawrence Hegarty, Joseph Yau and Waiming Wong. This development group[7] is supervised by Dr Peter Grogono.

The Dee group started this implementation project on May of 1991. The first Dee program which said **Hi folks, I am alive!** was successfully compiled, linked and executed on 21st December 1991.

[7]Dee group will be used to denote the people in the development team

1.4 About this thesis

This thesis is about the design and implementation of the Class Interface Manager, Linker and the Programming Environment.

In Chapter 2 and Chapter 3, the detailed design and implementation strategy of the environment were discussed. For this environment, we tried to implement our language and the programming environment in parallel. A class interface manager was introduced, which had many benefits for the system design and which facilitated the design of the programming environment. This approach is quite different from that of other researchers[LO90, Mey88]. As the result of our research, a strongly coupled and efficient system was built, each of the components of the system depending only on the *views* of the CIM.

In Chapter 4, a compact object record linker was discussed. We use a heuristic method to solve the problem instead of exhaustive search. Our CCIT is based on the CIT method. Both experiment and random testing show that our method is a significant improvement over CIT.

Chapter 2

Class Interface Manager

In the Dee programming environment, the class interface manager is an integral part of the system. The idea of using a database management system to maintain information in a software development environment is not new, but its effectiveness is limited if the components of the development environment are not carefully integrated.

Program development in object oriented languages typically requires more interaction with existing code and its documentation than does development based on other paradigms. The object oriented paradigm encourages programmers to extend existing classes by inheritance rather than to write completely new modules. Since the *inheritance* relation is more intimate than the *client* relation, programmers must acquire familiarity with the details of many classes before they can program productively. While programming, they will frequently need to confirm their recollections of the services provided by the classes they are using.

For these reasons, programmers who use object oriented languages need rapid access to accurate documentation about the software components they intend to use. Accurate documentation of a specified class can only be found in source code that has been accepted by the compiler.

Thus there is a need for storing some of the intermediate information captured or inferred by the compiler.

There is also a very important reason to store some of the verified information of a specified class. It is because all classes in an object oriented language are independent units which describe the abstraction or implementation of a class. The semantic analysis is a very important part of the processing of the compiler. This processing needs a lot of information about the other classes. All class information must be accurate and consistent as are created during compilation.

In the Dee programming environment, a class interface data base was designed to store class interface information and some information about the development system, such as class ownership, update time, interface change time, source location, etc.

The class interface manager was designed as a query interface of CIDB for other system modules other than the current module, for examples, the compiler, linker and the graphical user interface. The most important issue in building this layer was to enhance the integration and flexibility of the system and provide database maintenance facilities such as consistency and accessing control over the CIDB and avoiding fragmentation.

As Figure 1.1 shows, the compiler writes the interface of a class to the database after successfully completing the semantic analysis of that class. This is the only way in which the database is ever updated. The CIM provides information in response to queries. Both compiler and programmer can trust information they obtain from the CIM because the original source of the information is the compiler Information provided by the CIM is correct, consistent, and up to date. The CIM provides access to a rich repository of useful information about existing classes and Dee. In particular the CIM provides facilities and abilities for distributed software construction and management by maintaining the source code as private files and the database as a shared resource. In a natural way, programmers retain full control over their own classes while benefitting from classes developed by others.

2.1 The Class Interface

Dee is a strongly-typed, class-based, object oriented language which provides multiple inheritance facilities for both protocol and implementation.

The source text of a Dee program consists of a number of class definitions. Each class is defined by a single document called the *canonical document*[Gro91b, GC91, Gro91a] of the class. The canonical document is read and modified only by the *owner* of the class.

All the information about a class is contained in the *canonical document* of the class. A class is not visible to anyone other than its owner until it has been compiled. The compiler, in addition to semantic checking and code generation, constructs an interface for the class and writes it to the CIDB. Subsequent access to the class, by both programmers and the compiler, is made through this interface.

The canonical document of a class defines all attributes of the class. An *attribute* is either a definition of an *instance variable* or a *method*. Each attribute is either *public* or *private*. Clients of a class may use its public attributes but not its private attributes. Heirs of a class may use both its public and its private attributes. For example, a client of class *Part*, shown in Figure 2.1, could use *desc*, *satisfies*, and *show*, but not *cost*. The value of a public variable, such as *desc*, can be accessed, but not altered, by a client. The canonical document also specifies the classes which the defined class needs. The classes which the defined class inherits are listed explicitly. Other needs are indicated by declaration: for example, the declaration *desc:String* indicates that the defined class is a client class of *String*, the class of strings.

The canonical document may contain comments. In Dee programmers cannot write comments in arbitrary places because a comment is a terminal symbol in the grammar of Dee. For example, there may be a comment between two statements but not within a simple statement. In practice, the restrictions on the placement of comments are a minor inconvenience for programmers. The compiler writes selected comments to the class interface. Specification comments, which answer the question "What does it do?" are distinguished syntactically from implementation comments, which answer the question "How does it work?" A programmer who enquires about a method will see only its signature and specification documentation. Figure 2.1 shows the canonical document for an abstract class called *Part*. The document has been simplified for the purpose of illustration; typical documents would be much longer than Figure 2.1. A part has a description, a cost, and two methods. The method *satisfies* is abstract because it has no implementation in this class. The method *show* has an implementation and is therefore concrete. Since there is an abstract method, the class as a whole is abstract and can have no instances.

The class *Hook*, shown in Figure 2.2, inherits from *Part* and provides an implementation for *satisfies*. A programmer who wanted to inherit from *Hook* would see the view shown in Figure 2.4. The view contains the attributes inherited from *Part* as well as the attributes declared in *Hook* itself. The programmer would see the specification comment "Return true if the hook size satisfies the constraint" for the method *satisfies* but not its implementation comment "Use integer comparison".

```
class Part
    -- An abstract class describing an inventory part
    -- which satisfies a constraint.
inherits Any
public var desc.String
    -- Description of the part.
private var cost:Float
    -- Cost of the part.
public method satisfies (c:Int):Bool
    -- Return true if this part satisfies the constraint.
public method show:String
    -- Return a string corresponding to the part.
    begin
        result := desc + " " + cost.show
    end
```

Figure 2.1: Canonical Document for Class *Part*

```
class Hook
inherits Part
var size:Int
public cons make_hook (hook_cost:Float; hook_size:Int)
    -- Construct a hook with given cost and size.
    begin
        cost := hook_cost
        size := hook_size
        desc := "hook"
    end
public method satisfies (hook_min_size:Int):Bool
    -- Return true if the hook size satisfies the constraint.
    begin
        -- Use integer comparison.
        result := size ≥ hook_min_size
    end
```

Figure 2.2: Canonical Document for Class *Hook*

```
class Hook
ancestors Any Part
var desc:String
var size:Int
cons make_hook (hook_cost:Float; hook_size:Int)
method show:String
method satisfies (hook_min_size:Int):Bool
```

Figure 2.3: Client Interface of Class *Hook*

```
class Hook
inherits Part
uses Int String Float Bool
ancestors Any Part
desc:String
    -- Part descriptor
cost:Float
    -- Cost of a part
size:Int
show:String
cons make_hook (hook_cost:Float; hook_size:Int)
    -- Construct a hook with given cost and size.
satisfies (hook_min_size:Int):Bool
    -- Return true if the hook size satisfies the constraint.
```

Figure 2.4: Inherited View of Class *Hook*

The compiler, compiling a client of *Hook*, would see an encoded form of the information shown in Figure 2.3. The attribute *cost*, declared `private` in *Part*, is not visible to a client of *Hook* and is therefore not included in the interface.

2.2 Design issues

The most important design issues for CIM are: How is the CIM integrated with the other functional units? What should be the interface between the CIM and other components? What information needs to be stored? How much information can be re-created at run-time?

2.2.1 Integration and Interface

Figure 1.1 shows the relationships between the major components of the environment. The compiler, consisting of a scanner/parser, a semantic analyzer, and a code generator, is an integral part of the system, not an isolated component. This is the key to our approach, since it allows all components of the environment to benefit from semantic information derived during compilation. The abstract syntax tree, denoted by AST in Figure 1.1, is an intermediate data structure created by the scanner/parser and used by the semantic analyzer and the code generator. Class interface information is stored in CIDB, the class interface database. The diagram shows that all access to class interfaces is mediated by the CIM and that the class interfaces contain only information that has been semantically checked.

Figure 2.5 shows, as a service oriented functional module, the CIM provides four sets of query commands and *views* for the Dee compiler, linker and semantic browser[1]. The following is a summary.

- The response of CIM to a compiler query is in the form of abstract syntax tree.
- For the linker the view of output data is an syntax attribute list (SAL) which contains only the data of interest to the linker.
- For the semantic browser, the view of CIM is in form of textual data records.
- The interface between the graphical user interface (Dee Folder) and CIM are consisted of both textual data and syntax attribute list.

[1]Appendix A contains a detailed description of the queries.

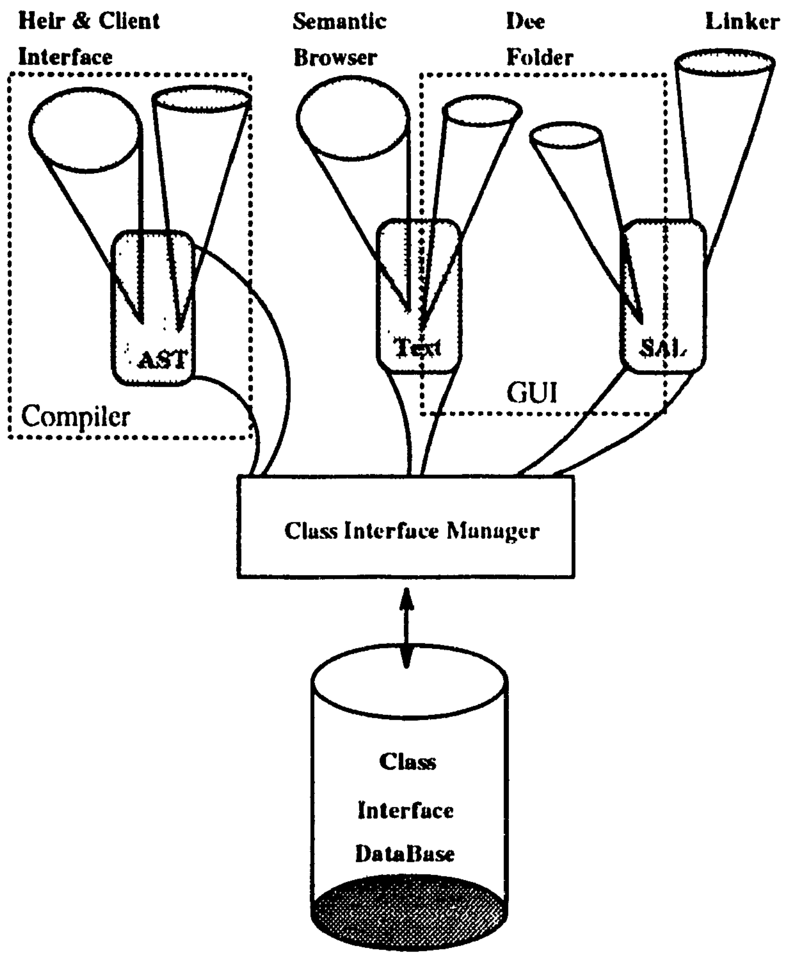


Figure 2.5: CIDB Data Image

The advantages of the design are: The CIM hides all the detail of implementation of the CIDB; The four interfaces can be seem as different *views* of the CIDB; Each view has its own functional aspect and is isolated from the others. For instance, the semantic checker of the compiler only need to knew about the interface is the functions provided by the CIM and that all data returned by the function are in the form of abstract semantic trees which the semantic checker can immediately use. The semantic checker need not to have extra functions to handle the output of the query.

2.2.2 Data

As described in the previous section, we have two kind of class interface for an individual class. They are: *Heir* interface, containing all the properties which could be inherited by descendants, and the *Client* interface, containing all the properties of an instance object which belong to that specified class. The *Client* interface is a subset of the *Heir* interface. So, the basic unit of data in CIM is a class's *Heir* interface.

For any given class's interface, we can decompose all the information into three categories:

- *Class Definitions:* Information includes class heading, *inherits*, *extends*, *uses* and *ancestors* information.
- *Instance Declaration:* Information about all the instance variables, including type of the instance and comments.
- *Method Signature:* Information including: method signature, comments and some extra information added by the compiler, such as: when it was declared, where it was implemented and which method categories it belongs to.

According to the requirement of linker, semantic browser and graphical user interface, CIM also stores some extra information about the classes, such as: last update time, source location, system environment variables, etc.

2.3 The Architecture of CIM

The current implementation of CIM is a interface between other system modules and the class interface data base system CIDB). The CIDB provides data storage and access to multiple user which was built upon the Unix file system with file locking facilities.

As the Figure 2.3 shows, the whole system was decomposed into 5 layers:

- **CIM service** is the query processing interface for other modules.
- **Parser and Packer** the parser is responsible for translating the data read from database into an abstract syntax tree. The packer is responsible for translating the abstract syntax tree into a logical text record which can be stored in the lower layer.
- **Database interface** provides a set of routines that can be used to read data from or write data to the CIDB file.
- **Database maintenance** routines are responsible for data update and transaction handling.
- **File handing** routines support low level file facilities.

This design has emphasizes the need for flexible system architecture in order to achieve the goals of software extensibility, reusability and compatibility. The

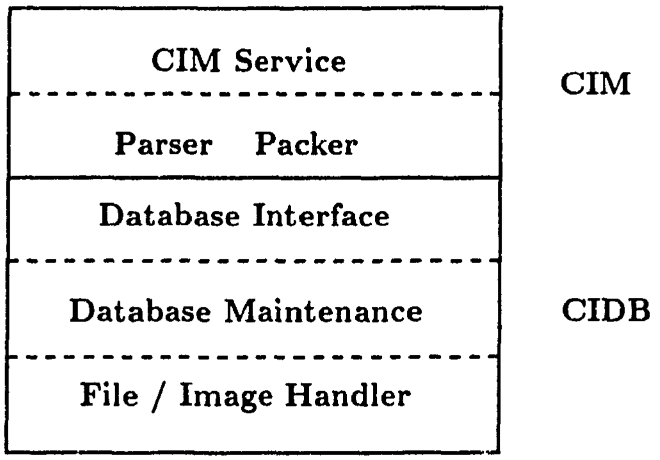


Figure 2.6: CIM System Organization

whole architecture reflects the basic techniques of modular decomposition. They are: *narrow the interface*, *weak coupling*, *explicit interfaces* and *information hiding*[Mey88, BR89]. Each module had been precisely designed according to its own functionality and usage.

2.4 CIDB

The CIDB is a small, fast data base management system. It provides the basic data base facilities to the upper layer. As a functional unit of language processing system, the CIDB has to address the major issues of its usage: store class interfaces, provide data retrieval facilities and control the consistency and integrity of all stored data.

For our application, the CIDB has to fulfill the following additional requirement:

1. First, it should be customized to the requirements of a compiler;

2. Second, it must carry out all queries within an acceptable time;

3. Third, the size of data stored in the data base should not be limited to fixed

size records, and it should allow varied logical layout;

4. Fourth, storage space should have an acceptable size;
5. Fifth, it should provide flexible record accessing method to support the mapping from AST to the data record pages(physical pages).

The first and second requirement is the key issue of design. They affect the overall performance of the system. The time and storage used by the CIDB must be acceptable to the computer and user. The third and fourth requirement reflects the aspects of the source text of the Dee program and the goal of the implementation. The fifth point addresses the criteria of software construction.

2.4.1 Data Storage Schema

The CIDB stores all data in the form of ASCII text. The whole data base can be seen as a memory image. It consists of binary index trees, textual data and a hash index table.

The binary tree is used to index names. There are two kinds of binary tree in the data base. The *class tree* contains all the classes in the system and the *attributes tree* contains all the attributes of an individual class. The reason we used a binary tree is not just for its simplicity, but also because it meets the requirements of object oriented programming. Most of the classes are typically small, consisting of no more than a few hundreds of lines of code. Each class usually has no more than 25 attributes. Thus the binary tree's search performance is acceptable for the systems for which Dee is intended to be used.

As the Figure 2.7 shows, there two different data units in the data base: Tree nodes and data records. Each tree node is a fixed size record consisting of indexes to

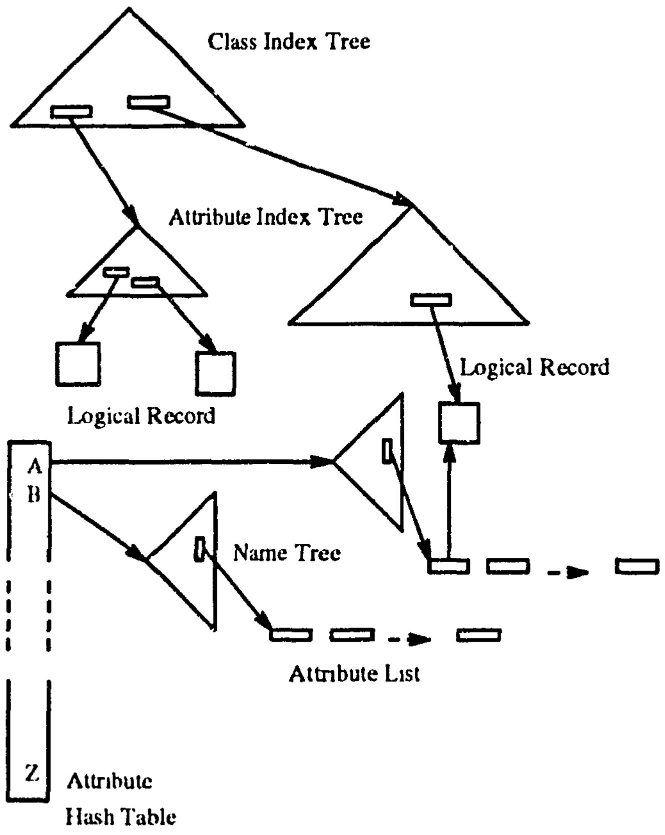


Figure 2.7: CIDB Data Organization

data records. There are two kinds of tree nodes: class node and attribute node; In the CIDB, the data record is the basic unit of stored data. It has a variable size and logical attributes.

The hash table in the CIDB is used to index the attribute binary tree of attributes that have the same name. As figure Figure 2.7 shows, it has 26 entries, each entry pointing to a binary tree which contain all attributes which have the same first character of its name.

2.4.2 Data record

Due to the requirement of the CIM, we implemented a compact, variable length for data storage. The data record in the CIDB is a logical unit. It consists of a field, a value and a sub-value.

A data record is a textual ASCII string terminated by a record mark. For any individual record, each field is terminated by a field mark. Each field may contain values and each value may consists of sub-values. Values and sub-values are separated by a value mark or sub-value mark.

The Figure 2.8 shows the logical layout of a record. This design provides several benefits for implementation, and both extensibility and compatibly for the system.

By using this design, the CIDB does not need to concerned with the layout of any given data record. For all data records, it needs to take care only of the length. It is clear that the logical field separation not only provides variable length and compact storage for the data records, but also provides the convenience for changing the record layout, for examples by adding a new logical member to the data record.

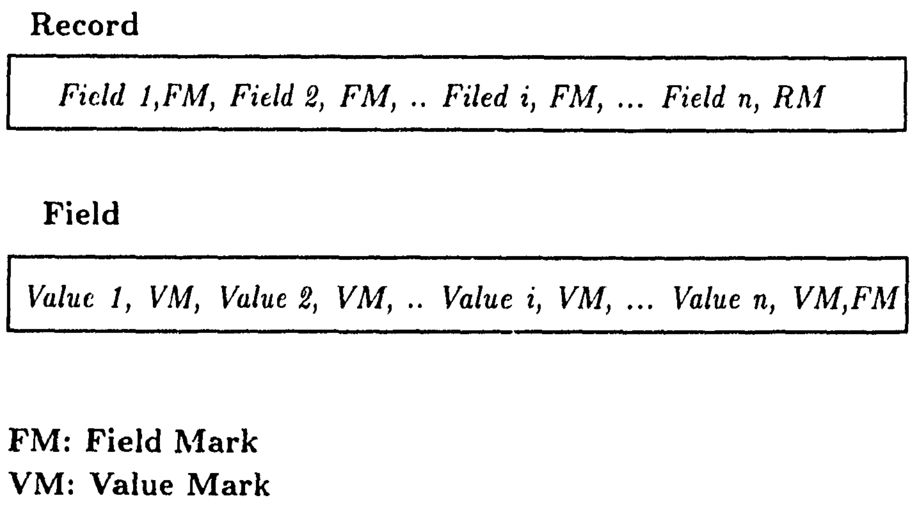

Record
Field 1,FM, Field 2, FM, .. Filed i, FM, ... Field n, RM
Field
Value 1, VM, Value 2, VM, .. Value i, VM, ... Value n, VM,FM
FM: Field Mark
VM: Value Mark

Figure 2.8: CIDB Data Record

2.4.3 Low-level storage format and Fragmentation handling

For performance reasons, CIDB maintains a special form of data in the image file instead of a conventional record file. This is because we need more flexibility and efficiency to build a language environment.

Image file

The CIDB stores all data in form of ASCII characters. The major reasons for this approach are: First, it is convenient for debugging in the implementation phase; Second, it increase the compatibility of the image file. All image files could be used on different machines and file systems.

The CIDB uses a special form of storage schema. It is different from a conventional data base system. As the Figure 2.9 shows, the whole image file is divided into two parts:

- *System cells region* is a reservation region for system information. The size

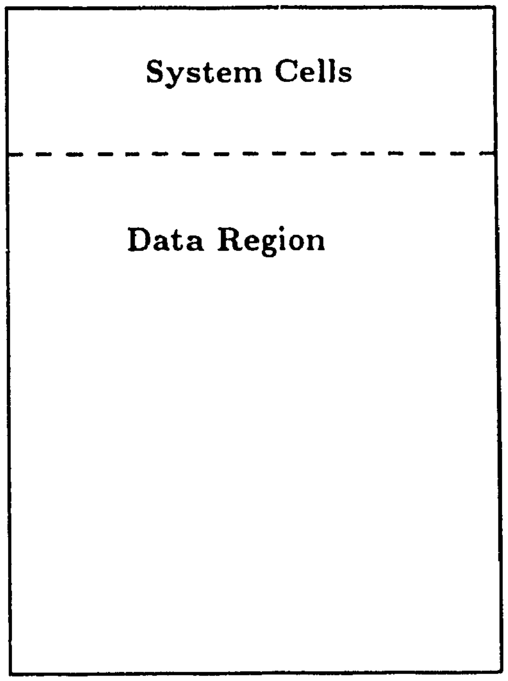


Figure 2.9: CIDB data File Format

of this region is 2K bytes, giving 200 slots for system information. Each cell slot would be used as a pointer to a special data segment which is in the data region, or used as a variable which denote the system status. The CIDB uses this region for storing dynamic information or special control state variables of the system.

- *Data region* is used to store all data, including class index, attribute tree, attribute hash table, data record and special data segments pointed by the cells in the system cell region.

2.5 Data Base Maintenance

For any data base maintenance system, there are two important implementation issues: fragmentation and accessing control.

Fragmentation is created by data updating. For instance, one technique for record updating is to delete the record first, then append the new one in the tail, then try to reuse the deleted region as soon as possible.

Because of the intended usage of the CIDB by the compiler, we implemented a very simple and efficient method to handle both issues.

According to the data flow of CIM, the only task which will update the data is the compiler. The semantic browser, linker and graphical user interface are read-only tasks. Moreover, the updating requested by the compiler is an atomic task updating the entire interface of the given class. This update action causes fragmentation, but the maximum space of fragmentation is known to be *total deleted space.* It close to the

$$\textit{number of delete request} \; * \; \textit{average size of interface.}$$

The fragmentation space depends on the number of delete requests. Thus, the CIM will perform a garbage collection after a certain number of delete requests. After experimenting, we set this figure to 3. Most of time, total fragmentation spaces are from zero to $10K$ bytes. The upper bound varies according to the size of the class's interface. For example, due to multiple inheritance, the standard classes of Dee have bigger size than the usual classes written by the user. The biggest class interface are *Float* with size $3.2K$. The upper bound may reach to $9.6K$.

Data access control of CIM is a very important issue for a multi-user environment. And also it is the technique invoked with transaction. These facilities are provided by the CIDB.

There two kinds of accesses action in the CIDB, data updating and data retrieval. Each update transaction should be an atomic task. During this task no other tasks are allowed to access the data base. For the CIM, the update transaction is the whole period of a *write* request. This *write* request needs to update the entire interface of the given class in one atomic task. Thus, the file locking control schema is suitable for CIDB.

When the CIDB receivs a *write* request, it will try to instantiate an exclusive lock on the data file. It allows the update only after successfully gaining the lock. All the *read* tasks share the data file at the same time.

2.6 Implementation Issues

There are currently three implementations of the Dee compiler and environment. Dee_1 was written in Turbo-Pascal for the IBM PC and compatibles by Grogono. Dee_2 is a revised version of Dee_1 designed and implemented by Grogono and Cheung. With other graduate students, we implemented Dee_3, a version which runs on Unix workstations.

The Dee_1 compiler writes class interfaces as small, separate files. Although the compiler must open and close many files while compiling a class, the speed of compilation is acceptable. Dee_2 stores class interfaces in a B-tree[2] indexed by class name and attribute name. Access to class interfaces is considerably faster with this organization, but the overall performance of the compiler is little better than with Dee_1. There are two reasons for this. First, Dee_2 was obtained by modifying Dee_1 rather than by starting afresh: efficiency is lost where new code interacts with old code. Second and more significantly, after compiling a class, the compiler rebuilds the B-tree index, which takes longer than we anticipated.

Dee_3 represents an attempt to combine the best features of Dee_1 and Dee_2. The

[2]We used the *B-Tree Filer* utilities to build the CIM

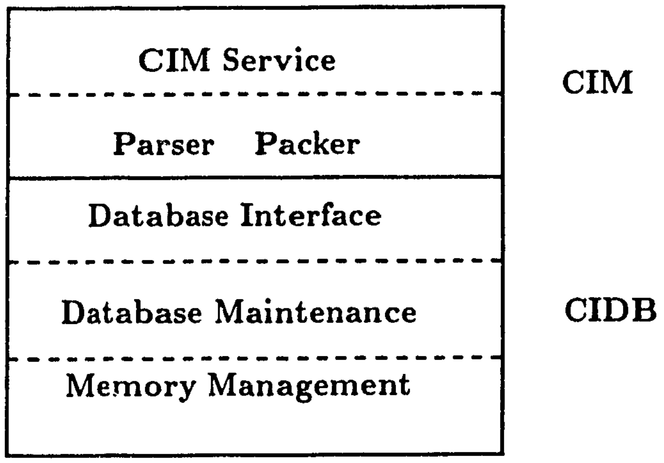


Figure 2.10: First Unix implementation: Memory Paging

first implementation of Dee$_3$ tries to achieve better run-time performance. It uses memory image concept, regarding the whole CIDB as real memory. During execution, CIDB swaps the data into heap space (real-memory) and maintain all data in several memory pages. This approach give an acceptable index update time and achieved a very high performance during run-time. Due to the need of concurrent accessing capabilities and support distribution development environment, a multi-user accessing technique was integrated into the second implementation of Dee$_3$. In this version, the CIDB is stored as a textual file, with the same hard image format as the previous implementation.

The current implementation of CIDB, has three image under its control. They are:

1. System standard image, it containing the basic class of Dee. For instance, *Any*, *Int*, *String*, *Float* and the classes of basic input/output classes;

2. Shared class image, it containing a set of interface of classes which may shared with others. For instance, a group of programmer works on a project, they

may have a shared project image. At this point, the class owners still have full control of their own classes, but the interface are public to others;

3. Private user image, it containing all users own classes.

All CIDB images have identical format. The CIM searches the class interface in the user image first, then the shared class image and system standard image.

2.7 Advantages of the CIM Design

The design has advantages for both the implementors of the development environment and the programmers who use it. For the implementors, symbol table management within the compiler is simplified. When the compiler requires information for type checking, it issues a query to the CIM.

One of the design principles of Dee requires that programmers should not have to provide the same information more than once. Another important principle is that all of the information about a particular entity should be in one place [Gro91b]. The canonical document and views support these principles: programmers are not required to write separate interface and implementation modules. In the canonical document, the definition of an instance variable or method consists of a single block of text; there are no export lists.

The CIM can determine whether a change to a canonical document changes the interface of the corresponding class. It can therefore decide how much recompilation is necessary and may even make fine distinctions, such as to recompile descendants but not clients.

The CIM maintains precise control over the content of a view. In particular, it can decide whether the information is needed for a client or for a descendant, and it reveals attributes accordingly. It may also reveal information to the compiler but not to programmers, for example the fact that a method is to be compiled in-line.

Programmers can trust information they obtain from the browser because the original source of the information is the compiler. Information provided by the CIM is correct, consistent, and up to date. The CIM provides access to a rich repository of useful information about existing classes. A browser need do no more than access this information in response to appropriate queries.

The Dee environment can support distributed development by maintaining the source code as private files and the database as a shared resource. In a natural way, programmers retain full control over their own classes while benefitting from classes developed by others.

Multiple versions of both individual classes and complete programs must be supported. With our approach, version control requires that the CIM maintain multiple versions and provide access to them as required by the compiler and by programmers.

The design of CIM addressed the requirement of flexible system architecture in order to achieve the goals of software extendibility, reusability and compatibility.

Chapter 3

The Development Environment

3.1 Introduction

Since the birth of computer languages, the only two tools we have used for programming are: a compiler and an editor. The compiler is used to generate the target machine code; the editor is used for editing our programs. Unfortunately, this has not been changed much for over 30 years, in spite of the fact that programming languages have changed from the first generation to the third generation.

In software development, programmers need rapid access to accurate and up-to-date information about the programs they are currently developing. And they need more tools to help them for their project development. For example, debugging tools, project management tools, source control, version control and system configuration management tools are all important.

An integrated programming environment is needed especially for an object oriented language which provide classes and inheritance, and in which programmers work by classifying all information into a hierarchy of related characteristics. Programs are in the form of abstract data types. Abstraction is captured in the classes hierarchy. Programmers need to know what classes can do and how they are related to one another.

Thus, it is very important for an object oriented language to have a integrated programming environment which can help the programmers to learn the current class hierarchy. The class hierarchy browser of **Smalltalk** is a good example, and becomes an important part of learning Smalltalk.

The Dee[1] system is an integrated programming environment. We have combined a strongly-typed object oriented language with an integrated, interactive development environment. For several reasons, we designed the compiler as an integral part of the Dee environment. Coupling the compiler and the browser simplifies symbol table management in the compiler. Conversely, the same coupling ensures that information is semantically checked before the browser displays it. Also, programmers do not have to understand the class hierarchy because the compiler creates class *views*.

The Dee environment, consists of: the **Dee Folder**, which is a graphical user interface for class hierarchy browsing and the query of class interface; the **Dee mode editor**, which is an editing mode of the *emacs* editor. It also support class interface queries and interactive editing facilities; the **Semantic browser** is a tool which can be used to make *ad hoc* information requests to the class interface management. All these facilities was built upon the CIM.

3.2 Motivation and Design Issues

The lack of reusability of software is a serious problem in the software industry. Object oriented programming has been proposed to solve this problem. By itself, the object paradigm is not sufficient. The solution should be an integrated and interactive software development environment. It must consist of language processing,

[1] Both the language and the environment is called "Dee".

project management, specification and documentation, self-learning and distributed development facilities.

The main motivation of Dee is to try find a better approach to reach our goal: An integrated, interactive development environment.

Biggerstaff [BR89] identifies four elements that would support software reuse: finding components, understanding components, modifying components, and creating components. A *software component* may be a library function, a module, or a class.

For the Dee environment, we also keep track of the following principles:

- The compiler should be an integrated part of development environment.
- Learning facilities must be provided. It must be easy to use and easy to learn the existing software components (classes).
- The knowledge captured by the interface must maintain consistency with the conceptual model of the language.
- Distributed development facility should be simple, easy to understand and straightforward.
- The ways in which programmer's work should not be complicated.

3.3 Dee

A significant aspect of the Dee environment is that we designed the whole system (compiler and the environment) at the same time and implemented them in parallel. In traditional language development, a team first designs a language, then writes a compiler for it, and finally goes on to develop a programming environment.

Program development in object oriented languages typically requires more interaction with existing code and its documentation than other paradigms. The whole development concept has been changed to classify abstract data types, find the objects, design class interface and arrange all classes into an inheritance hierarchy. The paradigm also encourages programmers to extend existing classes by inheritance rather than to write completely new modules. In particular, when the programmer wants to re-use some of the classes, he/she has to understand the details of the existing system and follow the existing abstract model in order to keep the extensions consistent with the current system. While programming, due to the dynamic binding, name overloading and their limited short-term memory, programmers will frequently need to confirm or to learn the services provided by the classes they are using.

For these reasons, typical object oriented environments support a *browser* which provides *views* of the class hierarchy and source code. The browser is usually syntactic, providing access to source text. The use of inheritance makes browsing facilities a very important tool for the programmer. At the same time it also complicates the design of environments which would help the user learn more faster and efficiently.

Object oriented languages have supported browsing since their beginning. For example, the **Smalltalk–80** programming environment allows programmers not only to inspect the class hierarchy, source code, and individual methods, but also can answer questions such as *What messages does this method send?*, *Who implemented this method.* [Gol84]. The **Actor** programming environment has the same kind of facilities. Specifically, user can ask questions like this *List all the sender who may be the implementor of the given method.* Since **Smalltalk** and **Actor** is interpreted, however, browsing retrieves only what the programmer wrote from the run-time system.

The browser does not provide any semantic information or any validated information. This is because, dynamic systems which interpret or compile incrementally do not usually perform static type checking.

Development environments for typed object oriented languages, such as C++ [ES90] and Eiffel [Mey88], are usually built after the language itself has been implemented and not as an integral part of the language design. These kinds of tools can only get information from the source code which may not be validated or may have semantic errors. In fact, many object oriented environments do not have any tools at all. Programmers are required to develop programs with no tools other than an editor, compiler, and debugger. The **Command Lisp and Flavor** object system is an example. Meyers reports that debugging an object oriented program under these circumstances can be difficult and frustrating because the information provided by the debugger is often not relevant to the problem [Mey90b].

In comparison, Dee has three major advantages: First, the information that the browser displays is correct and up-to-date and it is obtained during semantic analysis; Second, programmers do not usually need to be aware of the inheritance hierarchy, because each *interface* includes extra information inferred by the compiler; third, users can control the level of detail of the response to each query.

3.4 Dee Folder

The Dee Folder is shown in Figure 3.1, is a graphical user interface. It provides both the functionalities of the class hierarchy browser and the semantic browser. Users can browse through all classes related by inheritance, and can use the query facilities to make some specific queries.

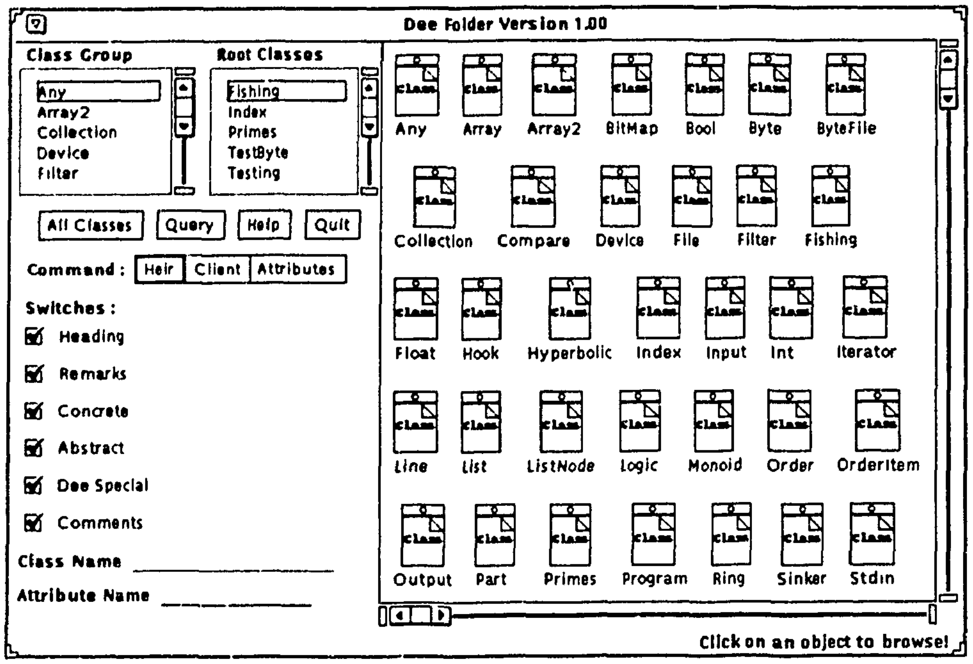


Figure 3.1: Screen layout of Dee *Folder*

3.4.1 Design Issues

As software systems grow in size and complexity, so does the difficulty of understanding and keeping track of relevant information. Research in software understanding attempts to mitigate this problem by creating useful views of both concrete and abstract aspects of software development. Visible, accessible and comprehensible information will enhance the understandability of software products.

Using simple and easy to operate graphical user interfaces that have become popular during the last ten years, the computer screen can be used to represent or simulate a lot of real work operations. For example, accurate pictures, or symbolic pictures (**icons**), represent objects which the user can manipulate by operations such as selecting, moving or activating a specified process.

The way of working with a graphical user interface is called **direct manipulation**, because it manipulates objects in a direct way. The major advantages are: first, object are visible, which helps the user to remember how to operate the objects; second, all manipulation has been transformed to basic work skills, such as *moving* and *pointing*.

Our goal in the design of the Dee Folder was to provide a graphically-based software visualization[Gol84] tool to the users. It can provide detailed views of individual software components and their derivations, as well as higher-level views of components and their relationships.

The major principles of our design of Dee Folder are:

- Provide a graphical user interface for class hierarchy browsing, semantic browsing and source code browsing.

- The interface should be consistent with our criteria of the design of our language: Simple and Easy to learn.

- The interface should capture and simplify the concepts of the object oriented paradigm. However, it should not mislead or confuse the users.

The main idea of design of Dee **Folder**, is to provide a clean and simple graphical user interface which must represent the nature of object oriented concepts and give users a tool to understand the development system.

3.4.2 The Information Provided

In order to help the user to have better working and learning tools, we have to provide a clean logical path from the interface to the users.

In general there two ways to understand an object oriented system. First, from an individual class, we can go through all related classes, identifying each class, going through the inheritance hierarchy, learning both the *heir* and *client* interface, then trying to learn the classes which have a *client* relationship with the current one.

Second, from the model of the system, we can learn the abstract concepts of each concrete class, then try to understand each individual class.

More often, the programmer will use the first method. It is the most straightforward, being based on logical thinking. In our experience, however, the second method is more effective than the first one. Using this way, the programmer will get the full picture and will have a basic understanding about the whole system. This is especially true when the class hierarchy is very deep and complicated.

Due to the nature of the object oriented technique, all the data are described in abstract and concrete classes. Genericity and dynamic binding give us the flexibility to re-use our code. In order to have better genericity and reusability, class designers will classify the entire real data model into an abstract inheritance hierarchy and make the interface of each class suitable for dynamic binding. As a result of doing this, the class hierarchy usually looks like an inverted tree. The topmost node is the most generic abstract class. The leaves of the tree are the concrete classes. From the top to bottom is the *view* of *global abstraction*; From the bottom to top is *local abstraction* which represents the hierarchical structure of an individual class. Both of them have captured the abstraction of the given data model, but in different *views*.

These two *views*, which we call *global abstraction view* and *local abstraction view*, have been incorporated into our user interface. We provide group by group, class by class, learning steps for the user.

The class interface is the most useful information derivable from a class. In Dee, the class interface not only contains the information provided by source text, it also includes some information inferred by the compiler. This information includes: *ancestors*, and *uses* set of that class; and all the properties that it inherits from its parents. Thus, for any individual class, programmers need not go through all the parents to get the full picture of the class which they are interested. Some special information also would be helpful for understanding and clarification. For example, the type of an attribute, where it was defined, and where it was implemented.

Figures 3.2 and 3.3 show the *Heir* and *Client* interfaces respectively of the class *File*. From this example, we can find that the interface includes all information about the class and it contains more information than the source code does.

From those two interface, we can tell that: the class *File* inherits from *Stream*, its ancestor set is {*Stream*, *Input*,*Output*, *Device*}; it may have instance variables or it will receive an object instance as the parameter of a method form the set of *uses* which is { *String*, *Int*, *Bool* }; the method *write* was defined and implement by class *Output*. The method *read* was defined and implement by class *Input*. The method *open* is defined by class *Device* but implemented by class *File* itself , and so on.

3.4.3 Layout Design

The layout design of the Dee *Folder* is intended to help users to learn our system.

Figure 3.1 shows, the layout of the *Folder*; The whole interface is separated into two parts, the control panel and the display window. There are no pop-up menu or pull-down windows.

The detail of the design of the panel is explained below.

Class List

There are two list windows on the left corner of the folder window see Figure 3.1. They are *Class Group* and *Root Classes*.

```
class File
inherit Stream
uses String Bool
ancestors Output Input Device Stream

public var handler:Int
     source class: Device
public var option:String
     source class: Stream public var path:String
     source class: Device

public cons assign (dev:String opt:String )
-- constructor
     Concrete
     Defined By : Stream; Implement By : File
public method close
     Defined By : Device; Implement By : Device
public method eof : Bool
--   return true if reach the end of file.
     Dee Instruction
     Defined By : File; Implement By : File
public method open --  Function open a file with its' option
     Dee Instruction
     Defined By : Device; Implement By : File
public method read (n:Int ) : String
     Dee Instruction
     Defined By : Input; Implement By : Input
public method readln : String
     Dee Instruction
     Defined By : File; Implement By : File
public method seekendof (offset:Int )
--   seek to a specified position from the end plus the offset.
     Dee Instruction
     Defined By : File; Implement By : File
public method seekoffset (offset:Int )
--   seek to a specified position from the current position plus the offset.
     Dee Instruction
     Defined By : File; Implement By : File
public method seekto (offset:Int )
--   seek to a specified position from the beginning of stream plus the offset.
     Dee Instruction
     Defined By : File; Implement By : File
public method write (buffer:String )
     Dee Instruction
     Defined By : Output; Implement By : Output
```

Figure 3.2: Heir interface of File

```
class File
inherit Stream

public var handler:Int
     source class: Device
public var option:String
     source class: Stream public var path:String
     source class: Device

public cons assign (dev:String opt:String )
-- constructor
public method close
-- close the file
public method eof : Bool
--  return true if reach the end of file.
public method open
--  Function open a file with its' option
public method read (n:Int ) : String
--  read n-type of date and return it as an string.
public method readln : String
-- read till end of the line.
public method seekendof (offset:Int )
--  seek to a specified position from the end plus the offset.
public method seekoffset (offset:Int )
--  seek to a specified position from the current position plus the offset.
public method seekto (offset:Int )
--  seek to a specified position from the beginning of stream plus the offset.
public method write (buffer:String )
```

Figure 3.3: Client interface of File

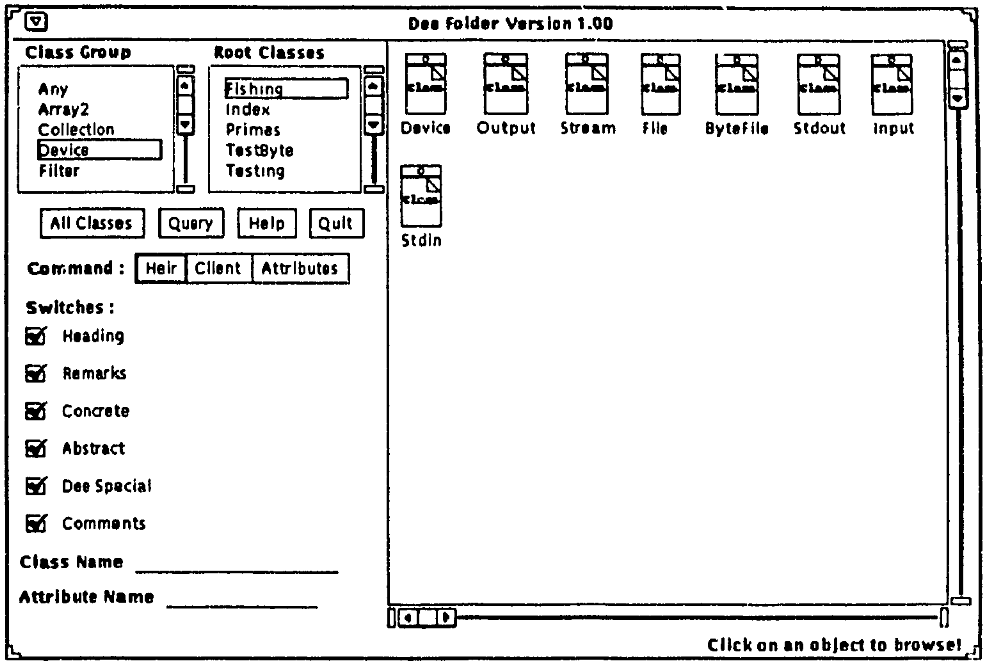
- **Class Group List** is the list of all classes that do not have parents. They are the classes which have the most generic form of abstraction and they are the topmost class of the class hierarchy.

- **Root Classes List** is the list of all root classes. All classes in this list have a method *entry*. They are executable classes , or complete program.

The **Class Group** list, is designed for capturing the concept of the *view* of *global abstraction*. When the user clicks one of the items on this list, all descendants of that node will be displayed in the graphical display window. By doing this, the user would get an idea of what classes inherit from the generic classes. As Figure 3.4 shows, all descendants of class *Device* have been listed in the display window. The user can click on the objects to see the detailed class structure of an individual class.

Basically, this list provides a facility for the user to do top-down scanning or searching. The graphical representation helps the user to remember the inheritance hierarchy by remembering the graphical pictures.

The **Root Classes** list contains all executable classes of the system. We try to capture the concept of *local abstraction* view by using this list. The users can click on one of them and get a graphical display of the class structure. Then, they can select any class they want to perform more queries. As Figure 3.5 shows, the class structure has been clearly displayed in the display window. On top of the class *Fishing* is its parent *Program*. All classes which have a *client* relationship with *Fishing* are listed beside it. Users can browse the class interface or the source code of *Fishing* by clicking the left or right mouse button. If they want to browse other classes, they can just point to another icon and press the left button, then the class structure of that class will be displayed in same manner as the class *Fishing*.



Switches :

✓ Heading

✓ Remarks

✓ Concrete

✓ Abstract

✓ Dee Special

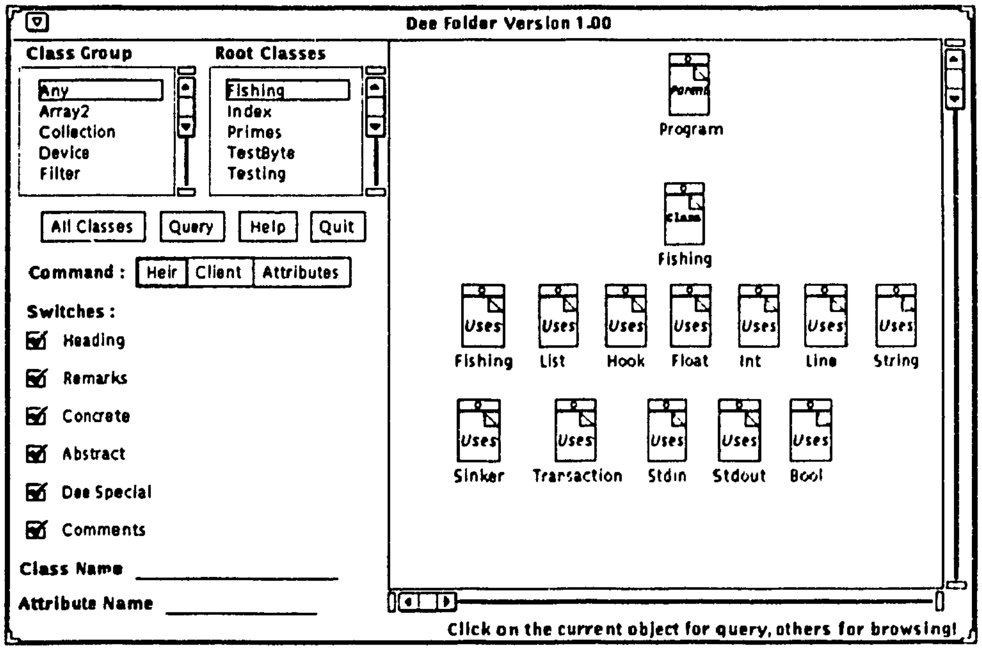
✓ Comments

Class Name

Attribute Name

Click on an object to browse!

Figure 3.4: Group classes of *Device*



Switches :

✓ Heading

✓ Remarks

✓ Concrete

✓ Abstract

✓ Dee Special

✓ Comments

Class Name

Attribute Name

Click on the current object for query, others for browsing!

Figure 3.5: Class structure of *Fishing*

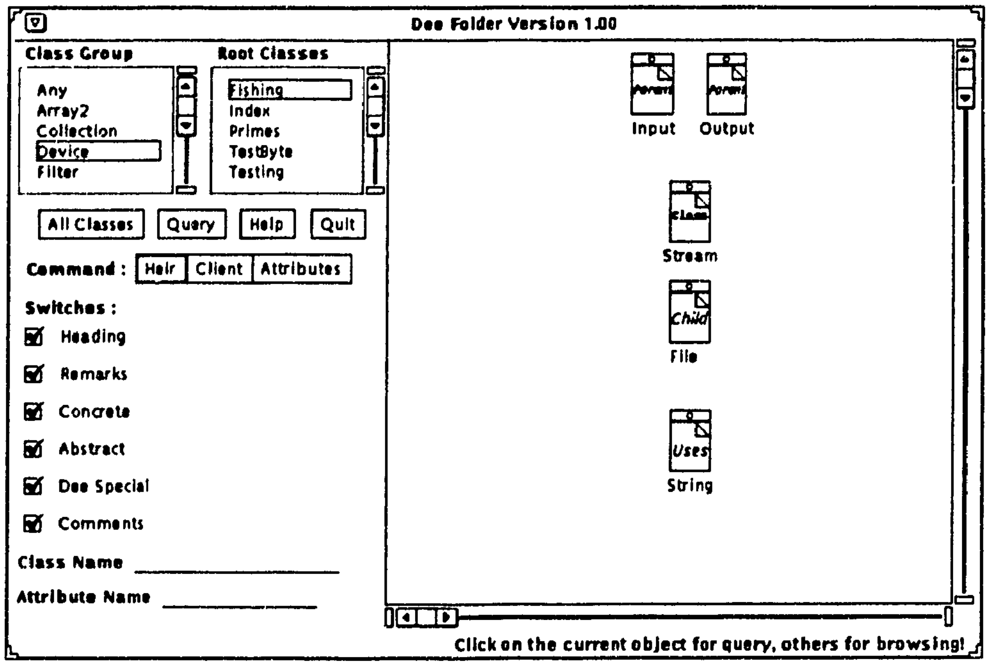


Figure 3.6: Class structure of *Stream*

Figure 3.6 shows, the class *Stream* has two parents *Input* and *Output*, one child, *File*, and it uses class *String*.

In general, this is bottom-up searching (from the root to the others). By doing this, user will get the full picture of all classes related to a particular class.

Buttons

There are two sets of buttons on the panel. The first set is for system commands. The second set is for query commands.

The command set button include, *All Classes*, *Query*, *Help* and *Quit*. The function of this button are listed below:

- **All Classes**: list all the classes in descendant order in the graphical window.
- **Query**: activate a query which specifies by query command and the query switches, and display the result in the graphical window.

- **Help**: active the interactive help system.
- **Quit**: quit the system.

The query button set is used to specify which interface user wants to query. The buttons *Heir* and *Client* denote that the query will be activated for *Heir* or *Client* interface respectively. The button *Attribute* denotes that the query is active for a specified set of attributes of a classes. For instance, the user can make a query such as *List all methods which have the prefix "str" in their name.*

Query Switches

The query switch on the control panel is used to control the level of detail of information which will be displayed.

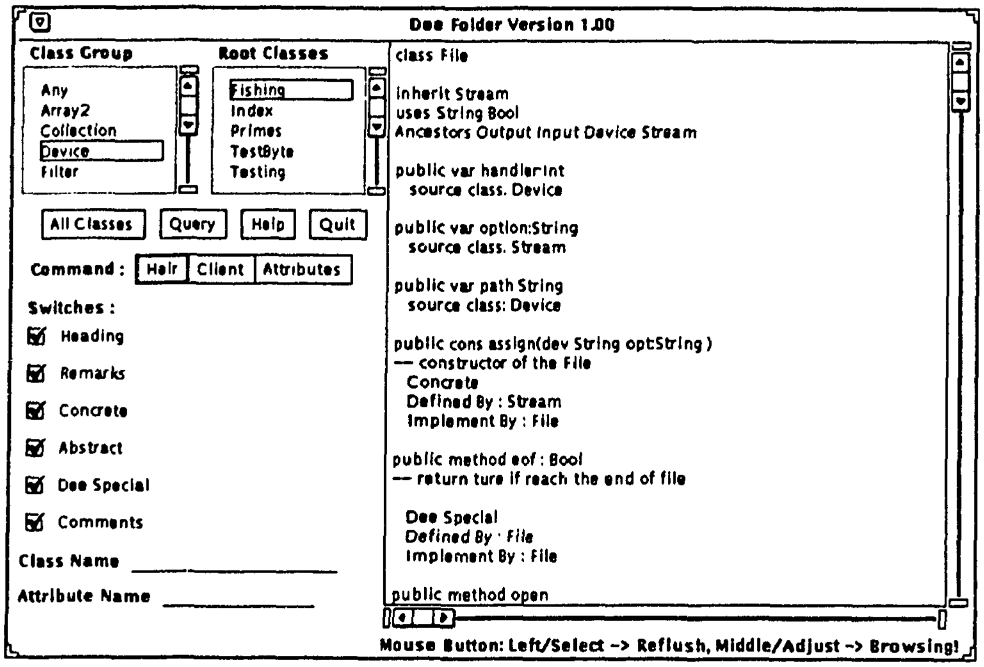
For each class interface, we can decompose all information into three categories:

Class Heading, including class definition, inheritance, ancestor, descendants, and uses information.

Attributes, in Dee, each attribute is either *public* or*private*. For methods, they may be *abstract, concrete* or *special*.

Extra Information, is the information inferred by the compiler or the comments of the attributes or the class.

Each query switch on the control panel is an on/off switch, denoting whether or not the user wants to display that kind of information. By the combination of the switches, the user can select the desired queries and control the level of the detail of the output. For example, they can ask queries such as: *List all concrete method in the*



Switches: ✓ Heading, ✓ Remarks, ✓ Concrete, ✓ Abstract, ✓ Dee Special, ✓ Comments

```
class File

inherit Stream
uses String Bool
Ancestors Output Input Device Stream

public var handler:Int
  source class. Device

public var option:String
  source class. Stream

public var path String
  source class: Device

public cons assign(dev String optString )
-- constructor of the File
  Concrete
  Defined By : Stream
  Implement By : File

public method eof : Bool
-- return ture if reach the end of file

  Dee Special
  Defined By · File
  Implement By : File

public method open
```

Figure 3.7: Query result: Full interface of class *File*

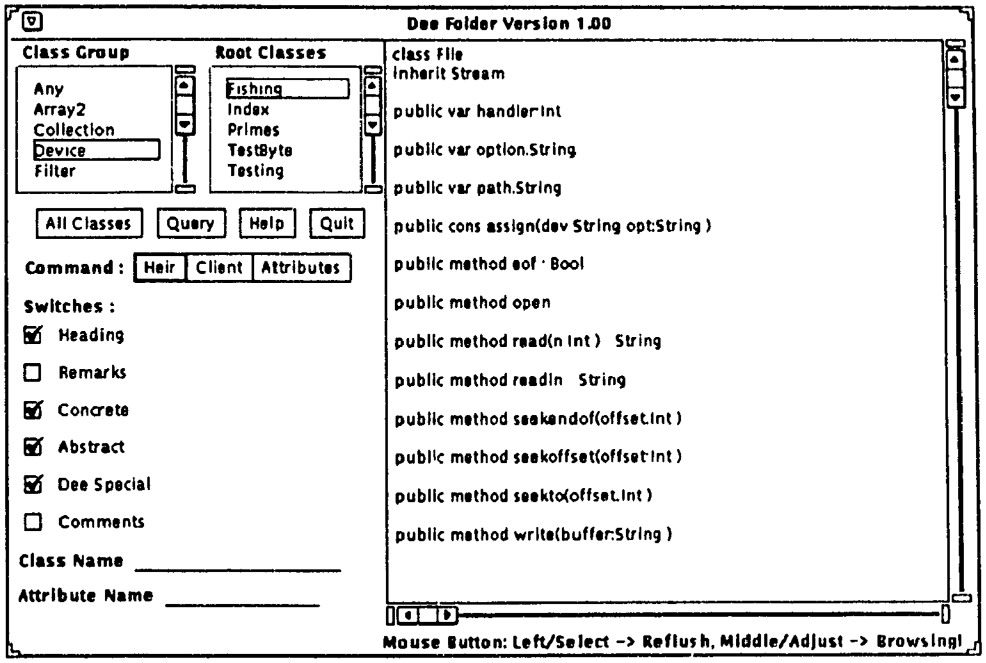
class String; *Which methods are implemented in Dee special*; *List all abstract methods of class* Comparable. The users can control the detail of the output by specifying whether they want to have comments and remarks.

Figure 3.7 and Figure 3.8 show the two different query results of class *File* created by selecting different switches. The user can get full information by selecting all the switches or get a summary short list by turning off the switches *Remarks* and *Comments*.

Icons

In order to give a better visualization to the users, the Dee *Folder* uses icon to represent graphical objects in the display window. Figure 3.9 shows all the icons used by the Dee *Folder*. They are:

- Class icon denotes the current class.



Switches : ✓ Heading, ☐ Remarks, ✓ Concrete, ✓ Abstract, ✓ Dee Special, ☐ Comments

```
class File
inherit Stream

public var handler Int

public var option.String

public var path.String

public cons assign(dev String opt:String )

public method eof · Bool

public method open

public method read(n Int )   String

public method readln   String

public method seekendof(offset.Int )

public method seekoffset(offset Int )

public method seekto(offset.Int )

public method write(buffer:String )
```

Mouse Button: Left/Select -> Reflush, Middle/Adjust -> Browsing!

Figure 3.8: Query result: Summary of the interface of class *File*

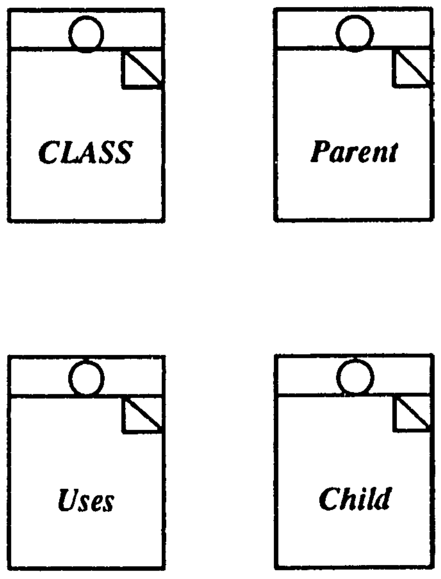
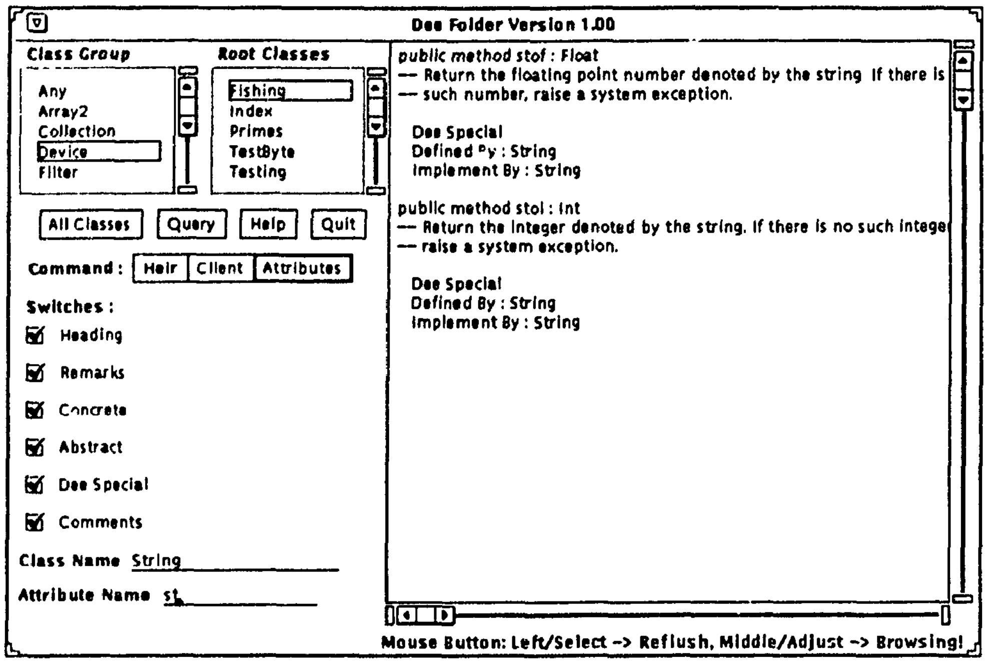


Figure 3.9: The icons of the Dee Folder



```
public method stof : Float
-- Return the floating point number denoted by the string  If there is
-- such number, raise a system exception.

  Dee Special
  Defined Py : String
  Implement By : String

public method stoi : Int
-- Return the integer denoted by the string. If there is no such intege
-- raise a system exception.

  Dee Special
  Defined By : String
  Implement By : String
```

Figure 3.10: Query: Find all the methods with the same prefix

- **Parent** icon denotes that this class is a direct parent of the current class.
- **Child** icon denotes that this class is a direct child of the current class.
- **Uses** icon denotes that the current class has a *uses* relationship with this class.

The mouse sequence

In a window environment, the pointing device, typically a mouse, is very important. It enables the user to interact directly with the system.

In general, as a pointing device, the mouse should simplify the interface by a well-designed event sequence. Otherwise, it will make the user confused about the operations provided by the iconified objects.

For the Dee *Folder*, we designed a very simple and straightforward mouse event sequence for the interface. In the current implementation, the users can just use the mouse to do most of the queries they want. A small number of queries require the

Class	Mouse Event	Function
Class Group	Left (Selected)	Display all the descendants in the graphical window.
Root Classes	Left (Selected)	Display the class structure of the selected item
Buttons	Left (Selected)	Perform the specified task.
Query Command	Left (Selected)	Reset the query command for the selected item.
Switches	Left (Selected)	On/Off

Table 3.1: **Panel: objects, events and its service**

user to enter the name of a class or attribute. As Figure 3.10 shows, users can make a query which displays all the attributes with the same prefix.

From the object oriented point of view, each graphic element on the panel or in the graphical window can be seen as a functional object which can receive a mouse event message and perform a specified action according to the received event message.

Table 3.1 shows all the definitions of the panel objects. Table 3 2 shows, all the definitions of the object in the graphical window.

Basically, users can do the browse, interface retrieval and read source code just using the mouse. Specially, for the graphical window, we can summarize the mouse event by the buttons: *Left* (select button), used for displaying class structure or activate a query to the item; *Middle* (adjust button), return to display the class structure from text browsing; *Right* (menu bottom), display the source code of current class.

Message

For any kind of interface, it is very important to have help or error messages. The help messages can let the user to learn and understand the operation, and also should provide hints when user need to know *what to do next.* The error message should clearly point out what is the error and what should be done next.

Class	Mouse Event	Function
Class Icon	Left (Selected)	If in list form, display the class structure; If in class structure, activate the query which specify by the panel.
Class Icon	Right (Menu)	If in class structure, browse the source code
Parent Icon	Left (Select)	Display the class structure of that class.
Child Icon	Left (Select)	Display the class structure of that class.
Query Text	Left (Select)	Refresh or Activate the query again.
Query Text	Middle (Adjust)	Go back to display the class structure.
Source Code	Middle (Adjust)	Go back to display the class Structure.

Table 3.2: **Graphical Window: objects, events and its service**

Nowadays, most user-friendly interfaces have interactive message to help the user to learn more quickly and easily. Most of the time users are likely to learn by trial and error.

The Dee *Folder* has both help and error message reporting. The error message is displayed at the left footer of the window with a beep bell. The help message are displayed at the right footer. The system will trace the users input and report what is the next correct action.

3.5 Semantic Browser

The Unix version semantic browser is a command line query tool for use with text-only terminal. It is a switch driven query command. Programmers can query different information by setting the value of an individual switch. Table 3.3 lists all of the switches. The semantic browser has same functionalities as in the Dee *Folder*. The only the difference is that the semantic browser can be active in any command window or a terminal window.

The command line format is as follows:

```
ri Class [Attributes] +/-[Switch]
```

Switches	Description
C	Display Client class information
D	Turn off the default switches set of current command
i	Inherited Attributes
a	Abstract Attributes
c	Concrete Attributes
d	Dee Specials
h	Class Heading
p	Public Attributes
l	Private Attributes
r	System Remarks
f	Full string matching for attributes

Table 3.3: **Switches of semantic browser**

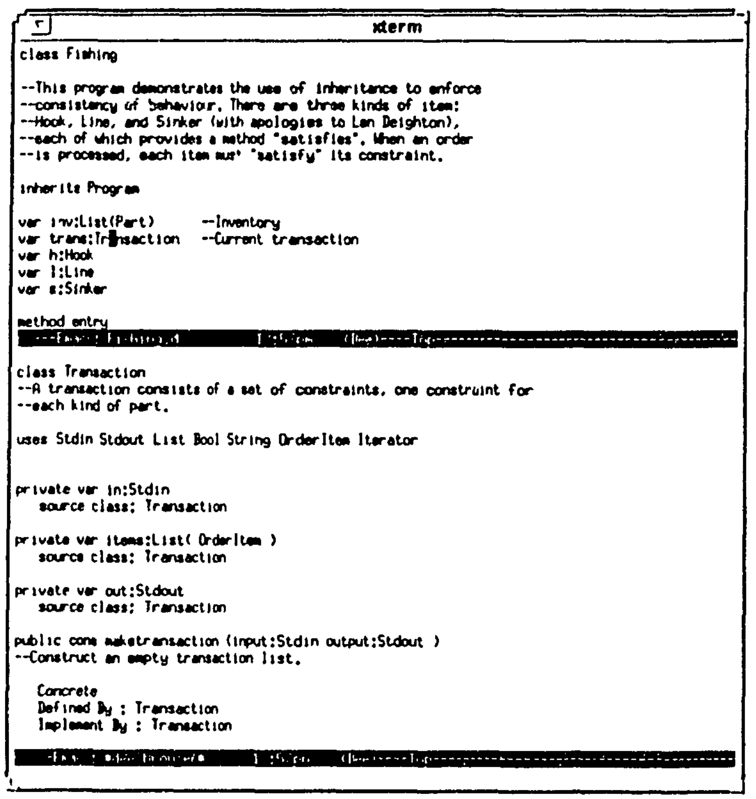
3.6 Dee Mode

emacs[Bar84] is a programmable editor. Due to its flexibility for modification, it has become the most popular editing tool in the Unix workstation environment.

We had chosen *emacs* for the Dee environment as our fundamental editing facility and we have re-programed the *emacs* according to our needs[2]. The detailed description can be found in the master thesis of Lawrence Hegarty[Heg92].

When the users run the *emacs* editor, if they had specified the file name with a extension "d", the *emacs* editor will switch to an editing mode (Dee Mode) which was designed for Dee. In Dee mode, an interactive editing facilities are provided for text editing. They include auto-indentation, template editing, and interactive semantic queries. Figure 3.11 shows that the user can uses the editor to interactively drive the semantic browser and get all the query output displayed in another window with a temporary buffer. This buffer and text will reside in the editor and the user can using editing command to read all of them.

[2]The Dee Mode for emacs was, was designed and implemented by Lawrence Hegarty In order to give a full picture of Dee, I have outlined its functions



```
class Fishing

--This program demonstrates the use of inheritance to enforce
--consistency of behavior. There are three kinds of item:
--Hook, Line, and Sinker (with apologies to Len Deighton),
--each of which provides a method "satisfies". When an order
--is processed, each item mus' "satisfy" its constraint.

inherits Program

var inv:List(Part)       --Inventory
var trans:Transaction    --Current transaction
var h:Hook
var l:Line
var s:Sinker

method entry
---Emacs: Fishing.d          [illegible]     (Eee)----Top------------------------------------------

class Transaction
--A transaction consists of a set of constraints, one construint for
--each kind of part.

uses Stdin Stdout List Bool String OrderItem Iterator


private var in:Stdin
   source class: Transaction

private var items:List( OrderItem )
   source class: Transaction

private var out:Stdout
   source class: Transaction

public cons maketransaction (input:Stdin output:Stdout )
--Construct an empty transaction list.

   Concrete
   Defined By : Transaction
   Implement By : Transaction

  -Emacs: [illegible]          [illegible]     (Eee)----Top------------------------------------------
```

Figure 3.11: Edit and Query

Interactive Keys	Command and Description
C-c	dee-class: get the class template
C-b	dee-browse-class: output to the browser buffer
C-B	dee-BROWSE-class: output to a new buffer
C-m	dee-method: get the method template
C-f	dee-function: get the function template
C-C	dee-cons: get the constructor template
C-a	dee-attempt: get the attempt statement template
C-i	dee-if: get if statement template
C-l	dee-loop: get for loop template
C-e	dee-elsif: get else-if template
C--	dee-comment-region: mark the whole region as comments
tab	dee-indent-line
return	dee-return
Del	backward-delete-char-untabify
ESC ;	dee-comment

C-x denotes that press Control C with x

Table 3.4: **Dee Mode Commands**

Dee mode also provides compiling and debugging interfaces. User can activate the compiler in order to compile the current editing buffer or activate the linker to link a root class. All error message from the compiler will be captured by the editor, and the user can uses a set of interactive editing command to trace through all the places which may have syntactic or semantic errors

Table 3.6 shows all the interactive key sequence and its functions.

3.7 Discussion

The Dee development environment is an integrated, interactive development environment. It provides class hierarchy browsing, semantic browsing, interactive editing, interactive compiling and distributed development capabilities for our object oriented programming language Dee.

Dee satisfies all four criteria identified by Biggerstaff[BR89] as fundamental to software reuse: finding suitable components, understanding components, modifying components and creating components.

Finding & Understanding components

The Dee Folder and semantic browser, provide both graphical and textual facilities enabling the user to find interesting software components. The user can use the Dee Folder to get a first impression or a quick overview of objects by following the inherit and client relationship of the objects. Specially, by integrating the editor and the semantic browser, an interactive *ac hoc* queries facility has been provided to our users.

The Dee Folder lists each class in the form of their abstraction relation instead of a long name list. User can quickly get to the point of the components in which they are interested. The iconified object representation not only provides visible information of the class but also enhances the understandability of the class hierarchies. Users will gain familiarity of other classes during browsing, because the browsing sequence of Dee Folder is the same as the class hierarchy.

The browsers in the *Smalltalk* and the *Actor*[3] programming language list all the classes in descendant order by the name of class. Most users feel confused at the very beginning. Both browsers provide three separate editing windows for class definition, instance variable and the method body. User will get confused about the class hierarchy and its structure. In [SB86] , it is stated that, users usually require at least 7 days to get familiar and comfortable with the user interface. When the system gets bigger, most of the users reported that they had difficulty understanding the full picture of the system.

[3]The author has industrial project development experience with Actor during 1989-1990.

One of the primary objective of the design principles for Dee is to provide a maintainable language to the users. The first design principle was that *information should not be duplicated.* The second was that *all information related to a program entity should be in one place.* These two principles directly contribute to the understandability of our system. The immediate consequence of the design are: Dee programmer do not write separate files for specification and implementations; All information about the class is in one place and there are no import or export lists in class declarations. Specially, the canonical document format separately defines the specification and implementation comments. Thus, the user interface is able to identify both of kinds of comments and use this information to help the development and system maintenance.

In Dee, programmers can choose the level of detail of the query output. They can easily make a specified query to confirm their knowledge or to recall the specification of an individual method or class.

Modifying & Creating Components

An interactive editing facility has been incorporated in Dee. In addition to these capabilities, it also has an interface to the semantic browser. During editing a program, programmers can directly make queries to the semantic browser if they need some information or help.

As Table 3.6 shows, besides the fundamental editing facilities, the Dee *emacs* editor, provides a set of custom-made functions. These functions give a lot of convenience to the programmer. For instance, by using the template retrieving commands, the programmer need not fully remember the syntax of the Dee language. They can get a template from the editor then modify it as they want.

A very important idea of the design of Dee *emacs* editor is: we try to give a complete development environment to the programmer. Programmers can do all the development work within the editor if they like. Compiling and linking facilities had been built with the editor. Specially, the error message of our compiler can be captured by the editor to help the user trace through all the errors in the source text.

Chapter 4

Linker

4.1 Introduction

A software system may consist of thousands of modules. We must be able to compile large programs one piece at a time. First, this avoids excessive compilation time. Second, separate compilation is a basic requirement of software construction and development. For class-based object oriented languages, the unit of compilation is at most a class, or preferably, a smaller unit such as a method. In Dee, the smallest compilation unit is a class.

One of the most important issue of compiler design is the run-time performance of the code generated by the compiler. Some of the commonly used techniques for code optimization include, peephole optimization, register allocation, loop unwinding and loop induction. An efficient and simple method of optimization, is to calculate all values that can be found at compile-time instead of finding them at run-time. For example, by translating identifiers to constant offsets, the compiler turns an expensive name search into a cheap addressing mode.

For any object oriented language providing multiple inheritance all the offsets determined by the compiler may be invalidated by subsequent inheritance. Either the compiler must resort to indirection, as in C++ [ES90], or offset calculations must be deferred until linking.

The Dee compiler [Gro91b, Heg92] translates a Dee class into a C language source file and saves all interface information into CIDB. The Dee linker uses CIM queries to obtain information about a set of classes or an individual class, then calculates all the offsets needed at run time. All this information is written in a C source file which defines the layouts of objects, class descriptors, a class table and the special objects. The C source files are then processed by a C compiler and a conventional linker to obtain an executable program.

For a strongly typed object oriented language run time efficiency is dependent on the mechanism of message dispatching. The problem is how to determine the correct method at run-time.

4.2 Run Time System

The Dee run-time system, shown in Figure 4.1, consists of a class table, class descriptors, an object stack and some special objects. The class table contains all classes for a particular Dee program. Each class is represented by a class descriptor. This data structure contains a list of parents, a method table, and the size and type of an instance of the class. The special objects include: *undefined object, false object* and *true object.*

At run time, each instance of object consists of a pointer to the class descriptor and *instance variable table*. Each slot in the instance variable table is a pointer to another instance which is an instance variable of that object.

The goal of the linker is to try to find out the value (offset of an instance object

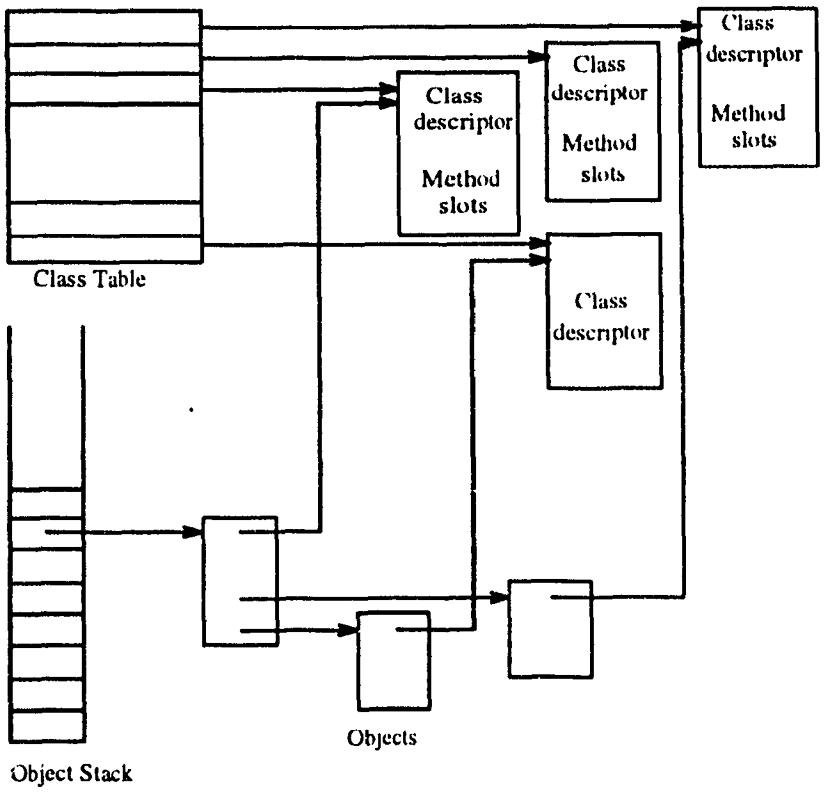


Figure 4.1: Dee: Run Time Environment

or address of the methods) of each slot in both the method table of class descriptors and the instance variable table of the objects.

4.3 Message Dispatching

The basic concept of message dispatching is very simple: an object will send a message to a specified object if it needs a service from that object. An object will perform a task pertaining to the received message.

In order to support this kind of facility, all object oriented languages need to implement a fast dispatching mechanism which will determine the correct code of a specified method of a class. At the conceptual level it is quite easy, but the difficulty is that we need to support dynamic binding in order to benefit from the object oriented paradigm. There are two major approaches to the problem.

- **Dynamic routine search:** At run-time, class descriptors are organized into a graph, mirroring the inheritance graph. This graph may be searched for the proper method. The searching usually starts at the receiver, continuing (if necessary) with all the parents of the receiver.

- **Constant routine offset:** At run-time, each class descriptor has a method table. An object linker will try to number all methods with a offset. The numbering scheme must respect the dynamic binding rules of the language. Basically, the rules are: all methods of a specified class should have a non-identical offset number and all methods which have same name in a class hierarchy should have the same offset number.

Dynamic routine search seems dangerously inapplicable when multiple inheritance is permitted. The whole inheritance graph may need to be searched, leading to unacceptable performance.

Most strong typed object oriented languages use the constant routine offset method to implement message dispatching.

There are three system components which contribute to this approach. First, the compiler performs a static analysis of a single class to determine what messages it can provide to others and what classes will be required during the run time. However, the compiler cannot determine exactly which methods it needs. Second, the linker needs to analyze all of the compiled classes required by a root class. The linker has access to more information than the compiler but still cannot determine exactly which methods will be required at run-time. Finally, the run-time system needs information which can be used to determine the correct method.

Here is a definition of the problem in a more abstract way. The function f computes the address P of the code of the required method. N is the name of the method, and the class of the receiver is C. In symbols,

$$P = f(N, C).$$

The method name, N, is known at compile-time, but the class of the receiver, C, is not known until run-time. (Hence the term "dynamic binding.") The problem is to compute $f(N, C)$ efficiently.

In Dee, the linker constructs class descriptors for all classes which are needed by a specified root class. Each class descriptor consists of a method table, inheritance information and some special information for the run-time system. For instance, the size of each instances of classes, the inheritance relation between classes and run-time status of the object. The dispatching mechanism is: locate the receiver first, then its class descriptor, and finally the method. Thus, this dispatching mechanism can be directly transferred to the following C code.

$$\textit{receiver} \rightarrow \text{class descriptor} \rightarrow \text{method table[method offset]} \qquad (4.1)$$

This solution is used in various forms by most typed, object oriented languages. Its time performance is acceptably efficient, but the space usage (size of the method and instance variable table) depends on the linker's algorithm which assigns the offset of the attribute and method table.

The simplest solution is, the run-time system maintain a method table which is a two dimension array, denoted as $T(C, N)$. The x axis is the class name, C. The y axis is the *method name*, N. The function f will be:

$$f(N,C) = T(C,N).$$

Obviously its time complexity is $O(1)$, and space $O(|N| \times |C|)$, where $|N|$ is the number of method names and $|C|$ is the number of classes.

From this point, we could improve the time performance and space requirement by using the following techniques:

1. determining cases in which dynamic binding can safely be replaced by static binding;

2. compressing the method offset tables.

The first technique is used for improving time performance by eliminate the unnecessary overhead of function calls. The second is mainly focusing on improving the space requirement. The current implementation of Dee is mainly focusing on the space issue.

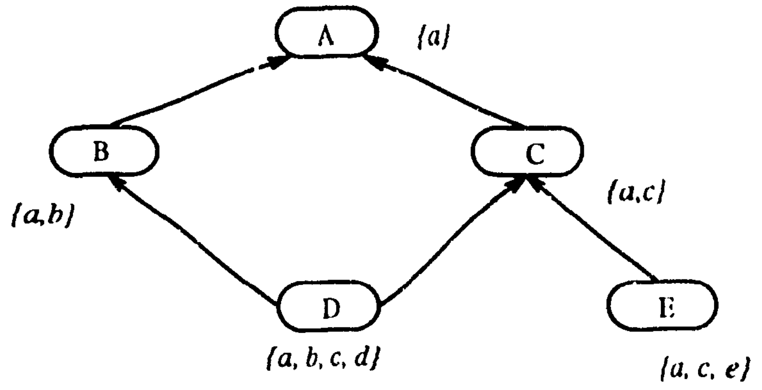


Figure 4.2: Color Example

As the code fragment 4.1 shows, the dispatching mechanism of constant routine offset depends on the method tables of each classes and the value of method offset (selector). The size of the method table is equal to the maximum value of the method offset of that class. Thus, if we can minimize the value of method offset of each class, then we would get a optimal space consumption at the run time.

4.4 Coloring Methods

In general, finding the offset of a method in the method table is quite easy. We can transform this problem into a graph coloring problem.

Figure 4.2 shows that A, B, C, D and E are classes. Class B and C have a single parent A. D has two parents, B and C. Class E also has a single parent, C. Assume that each class defines a single attribute. The attributes are a, b, c, d and e, respectively.

The rule for assigning the attribute offset is: no two attributes in any class have the same value. Thus, we can transfer the graph $G = (V, E)$ in Figure 4.2 and its relation between the attributes into a conflict graph which is illustrated in Figure 4.3.

Let $CG = (V', E')$ be a conflict graph. Each vertex $v' \in V'$ is a attribute. The

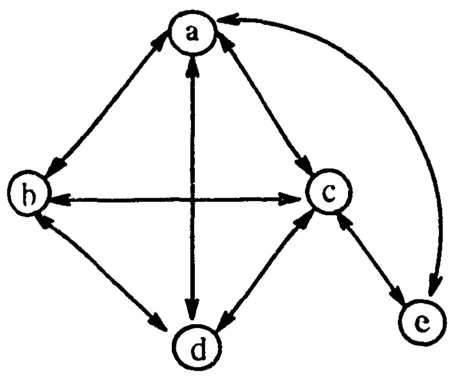


Figure 4.3: The conflict graph of the attributes

pair (u', v') belong to edge set E' if u' could not have same value as v'.

At this point, we find that the offset value of each attribute can be easily found by coloring the conflict graph, such that, there are no two connected vertexes with the same color.

This general technique is called sequential coloring. First assign a color to a node and then pick one node after another and assign it the minimum color that does not conflict with any adjacent node which has been already colored. This simple technique will find a minimal coloring if the nodes are chosen in the right order.

4.5 CIT

The color indexed table (CIT)[DMS89] technique is an implementation which uses the coloring method to assign the offset of attributes.

In order to get a closed minimal coloring result, the **CIT** technique sorts all the nodes in the conflict graph in descending order on the number of nodes adjacent to the given node (degree).

By using **CIT** technique, the size of the table associated with each class is reduced from *number of attribute names* × *number of classes* to a much smaller number, close

Class	CIT-best	CIT-worst	Optimal
A	1	5	1
B	2	5	2
C	3	5	3
D	4	5	4
E	5	5	3
Total Space	15	25	13

Table 4.1: **Space consumption of CIT method**

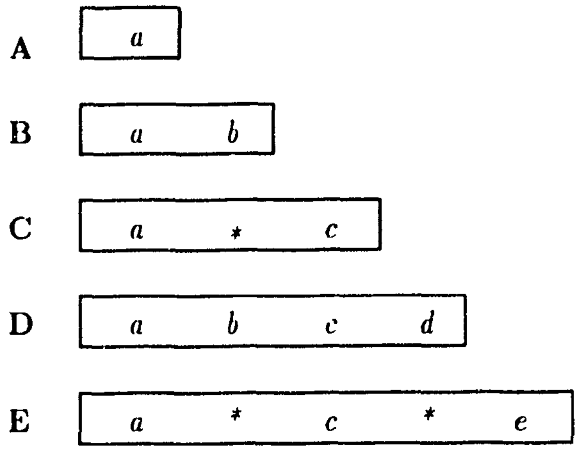
to ($|S| \times |C|$), where $|S|$ is the number of attributes of the biggest class and $|C|$ is the number of classes.

Figure 4.4 is the result of the **CIT** method. The weakest point of this technique is that the degree of the nodes of the conflict graph does not represent the real class hierarchy, and the results of the coloring are very dependent on the coloring sequence. Table 4.1 and Figure 4.4 show, the worst and best coloring. In the best case, space consumption is 15 with 3 empty slots. In the worst case, space consumption is 25 with 13 empty spaces: ''*'' denotes an empty slot.

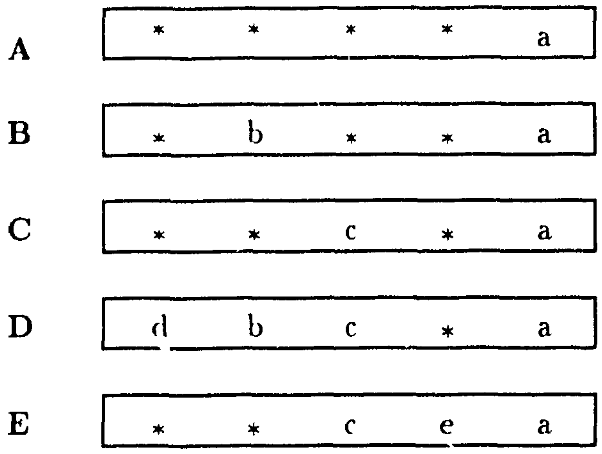
Figure 4.5 are the optimal coloring result. Table 4.1 shows the comparison between them. The total space consumption for the worst case of (CIT) is ($|N| \times |C|$), where $|N|$ is the total number of attributes and $|C|$ is total number of classes.

4.6 The Heuristics

Our compact coloring indexed table (CCIT) technique is based on the CIT method. In order to improve the coloring result, Dee uses heuristics,described in the next section, to arrange the coloring sequence which more closes to the class hierarchy and tries to get a densely packed attribute table.



(a) Best Colours



(b) Worst Colours

Figure 4.4: Result of CIT method

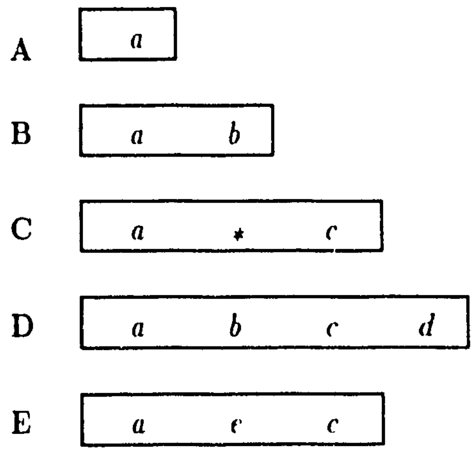


Figure 4.5: The optimal coloring result

4.6.1 Sequence

For a given set of classes, the size of its attribute table is the maximum color of the attributes; The best case is when the number of attributes is equal to the size of table. As Figure 4.2 shows, empty slots will be added to the table according to the inheritance structure. Different coloring sequences produce different results. The major difficulty is to find the sequence which will produce the optimal solution. The determination of a coloring of minimum number is a computationally difficult problem. Finding a coloring which has near the minimum number of colors is quite easy.

First of all we need to analyze the class structure appropriate to object oriented programming development.

For any strongly typed single inheritance language, like C++, the coloring problem is simpler than with multiple inheritance. With single inheritance, all class hierarchies are in form of a tree. The dynamic binding can only perform on a single branch. Thus, there will be no empty space if the coloring sequence is from the top-most node to the leaves. For multiple inheritance case, the classes hierarchy is more

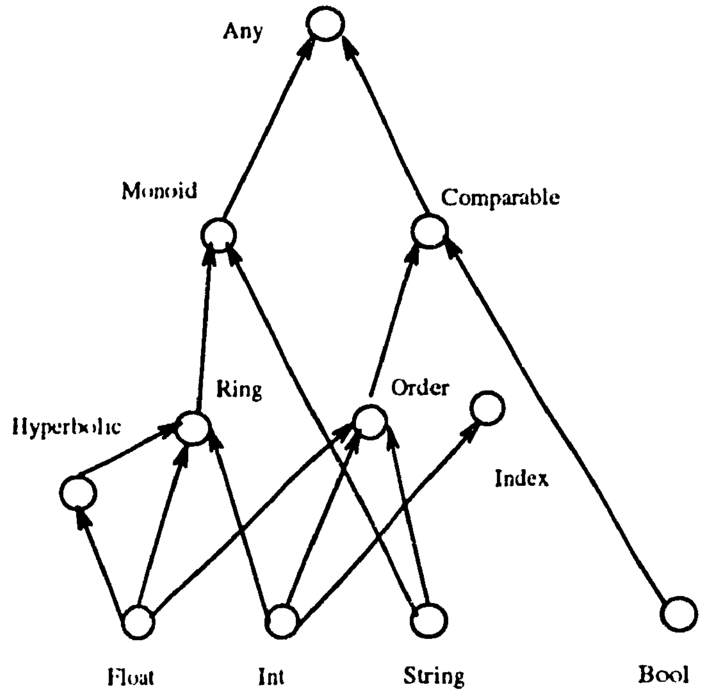


Figure 4.6: Basic class of Dee

complicated, it could be any kind of graph.

In the object oriented paradigm, we encourage programmers to reuse and make extensions to the existing code rather than rewrite from scratch. The most important aid for reusing code is inheritance. But reuse depends on the class hierarchy of existing classes in which we encapsulate the real data model. Most often the class designers will try to classify all the real data model into several class groups according to their abstract or functional aspects. Each group is encapsulated with different conceptual data structure or real objects. Then the class designers arrange all classes in one group into a class hierarchy which would lead a well-defined and easily reused conceptual model.

By doing this, the class hierarchy of an typical object oriented software system will consist of several groups of abstract data type. In order to get better reusability and extendibility, each of this group will be constructed into a tree-like structure.

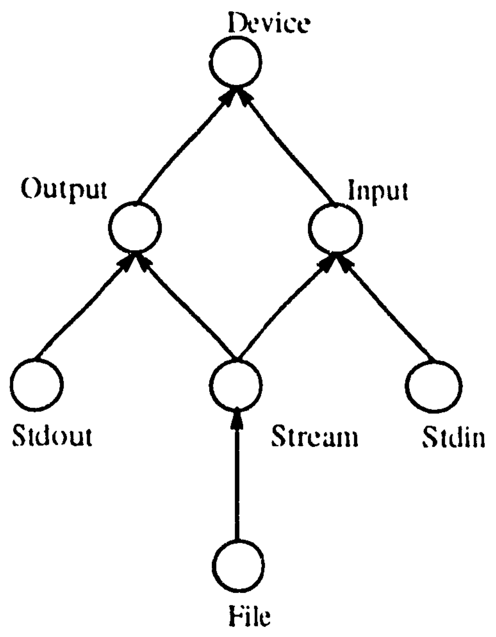


Figure 4.7: Classes for input/output

The topmost nodes of the inheritance tree are the generic class or abstract class. All the concrete classes[1] are the leaves of the inheritance tree.

Figure 4.6 and 4.7 show the basic abstract data groups in Dee: Figure 4.6 shows the basic classes: they are **Int**, **String**, **Float** and **Bool**; Figure 4.7 shows the basic Unix input/output devices: they are **StdIn**, **StdOut**, and **File**.

In Dee, even the simplest root class will probably use these two abstract data groups. Figure 4.8 shows, the relations between groups are *client* relations.

Through observation, we found that, for any concrete classes in the graph, we can build the class hierarchy graph in direction of bottom up. As Figure 4.9 shows, Each graph includes all the properties of that class. The size of indexed table is the total number of its attributes. If a class have more than one parent's, there will be empty space in the parents descriptors. The way to deduce the empty space in this tree is to color the whole tree from the topmost generation to the lowest one. If there is

[1]A concrete class does not have any abstract methods

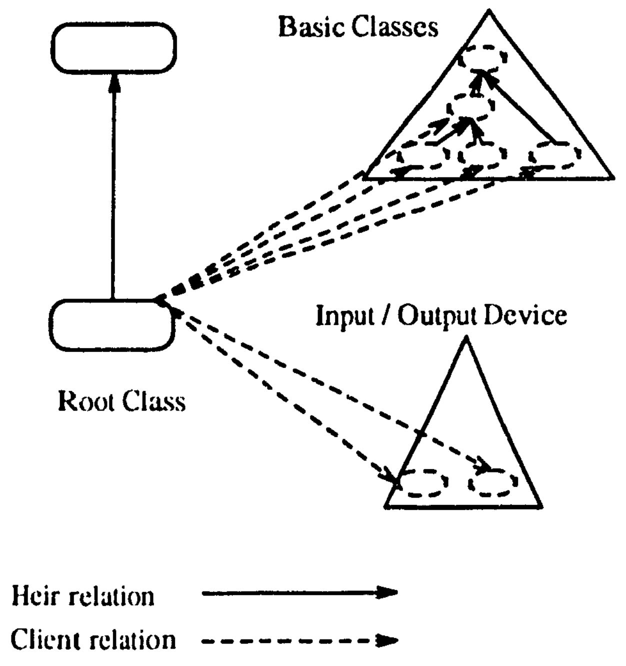


Figure 4.8: Relations between abstract group

more than one classes in the same generation, we should choose the class which has the smallest number of attributes.

4.6.2 Packing

Figure 4.2 illustrates that at most one of the descriptors of B, C can be densely packed. If C is not densely packed, then the descriptor for class E can make use of one of its empty slots, as shown in Figure 4.5.

Because of the simple nature of this example, there is only a small advantage to be gained by the compact packing technique, but it illustrates the point of our compact color indexed table.

4.6.3 The Algorithm

It is clearly infeasible to obtain the optimal coloring of offsets by exhaustive search. Instead, we use heuristics to find a solution which is not too far from optimal. Since

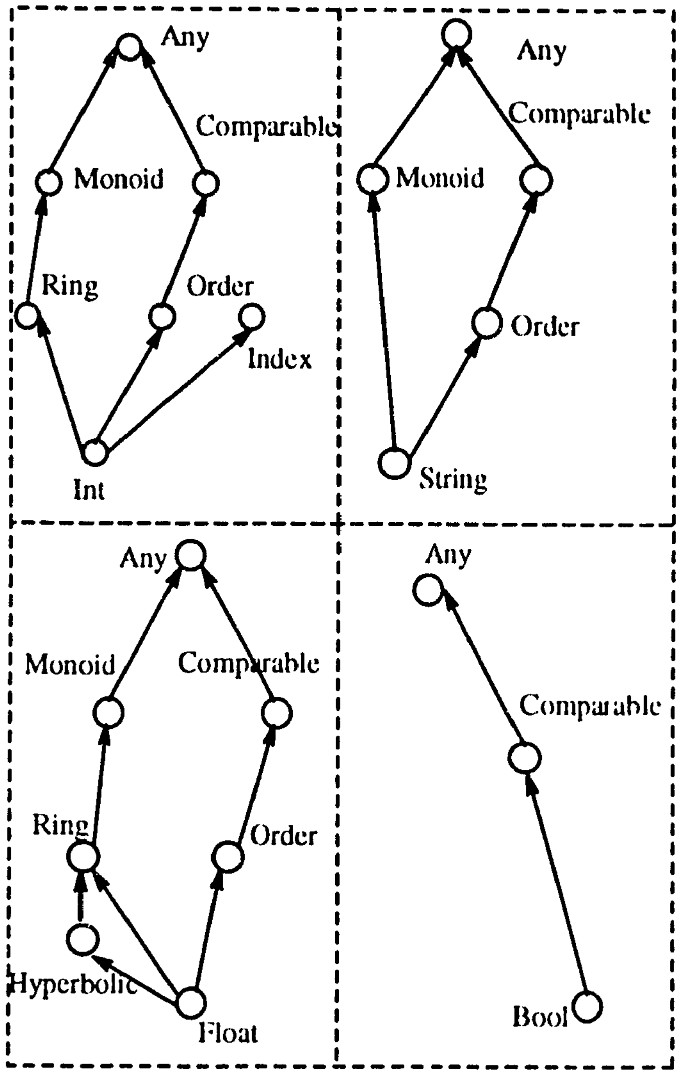


Figure 4.9: Decomposed sub-graph of basic classes

the problems of coloring offsets to instance variables and methods are similar, we do not need to distinguish between them.

The coloring algorithm must choose colors for the attributes in such a way that no two attributes in any class have the same color. The Dee linker performs the following steps:

1. Starting from a given root class, construct the set S of all classes needed by the root class. (*Need* is the transitive closure of the union of the *client* and *heir* relations.)

2. Construct the inheritance graph $G = (V, E)$ of all the classes in S. Each vertex $v \in V$ is a class. A pair (u, v) belongs to the edge set E if class u inherits from class v.

3. Find the sets U' and V', such that $U' = \{u | u \in U \wedge v \in V \wedge (u, v) \in E\}$ and $V' = \{v | u \in U \wedge v \in V \wedge (u, v) \in E\}$.(Informally, U' is the set of classes which have parents; V' is the set of classes which are parents.)

4. Find the set R, such that $R = U' - V'$. For each source $r \in R$ construct a subgraph $G_r = (V_r, E_r)$ containing only u and vertices reachable from v. (Informally, for each class which has no descendants, construct a subgraph containing that class and its ancestors.)

5. Color each subgraph G_r in turn, starting with the subgraph that has the smallest weight and proceeding in order of increasing weight. The "weight" of a subgraph is the total number of attributes (methods and instance variables) that it contains, not the number of nodes.

Class	#. Attributes	Biggest color	Empty space
Testing	6	6	0
File	14	14	0
String	24	24	0
Int	25	25	0
Stream	7	7	0
Bool	7	7	0
Float	33	33	0
Input	5	5	0
Output	5	6	1
Any	1	1	0
Device	2	2	0
Total	129	130	1

Table 4.2: **The Minimal Coloring Result**

Coloring Subgraph

For each subgraph, the Dee linker tries to color it with dense packing, using the following strategy.

1. Chose the node v in the graph G with oldest age and least *weight*. For nodes in the same generation, choose the one with smallest *weight*.

2. Color all attributes of the given node with the smallest colors, trying to re-use the empty space these create by adjusting sub-graph as soon as possible.

4.7 Results

The Figures 4.6 and 4.7 show the two inheritance graphs generated for programs that do not introduce any additional inheritance. Table 4.2 shows, the descriptors generated by the linker for these programs have only one empty slot, in class **Output**.

Since the Dee project is still in its infancy, we do not yet have larger than 40 classes programs to test the linker.

Classes	Attributes	Empty Space Ratio	Max colors
50	$125 \sim 656$	$0 \sim 26$	$6 \sim 38$
75	$287 \sim 1195$	$1.9 \sim 17.5$	$12 \sim 61$
100	$413 \sim 2374$	$6.7 \sim 17.1$	$9 \sim 74$

Table 4.3: **Random testing with inheritance 0 to 3.**

Classes	Attributes	Empty Space Ratio	Max colors
50	$123 \sim 1448$	$7.1 \sim 42.1$	$6 \sim 78$
75	$311 \sim 2536$	$5.1 \sim 43.3$	$8 \sim 91$
100	$289 \sim 2968$	$5.31 \sim 53.1$	$8 \sim 95$

Table 4.4: **Random testing with inheritance 0 to 4.**

However, we have experimented with some randomly generated test data. The Table 4.3 and Table 4.4 are two examples: For each table, the random model tries to generate similar inheritance structure with different number of attributes and distribution of attribute over that class structure. The maximum inheritance of Table 4.3 is 3 and of Table 4.4 is 4. The figure $a \sim b$ denote that the specified value are form a to b.

All random tests yielded a result with minimum color for the biggest table. Through the observation of the performance of random test, we could not find a linear relation between any of the following pair, {*empty space ratio, total number of attributes* }, { *empty space ratio, maximum number of inheritance* } and {*empty space ratio, schema of distribution of attributes* } . This is due to the nature of object oriented programs. The total empty space is dependent on all three of them.

These technique improved the *CIT* method, the worst case for the **CIT** and **CCIT** techniques is illustrated in Figure 4.10. There are N classes, and M attributes. All the attributes are distributed on the $class_2$ to $class_{N-1}$, and $S = \frac{M}{(N-2)}$ is the number of attributes of those classes.

For **CCIT**, the coloring sequence will be from $class_2$ to $class_{n-1}$. $class_2$ will

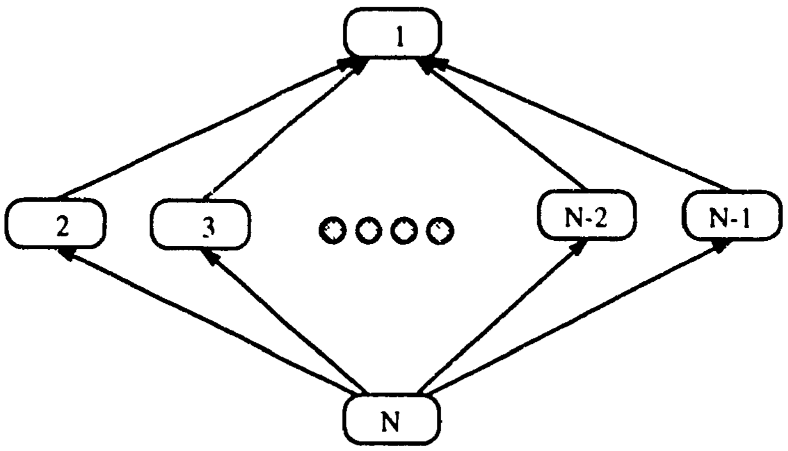


Figure 4.10: The worst case of CIT and CCIT

create S slots in $class_3$, $class_3$ will create $2 \times S$ slots in $class_4$, and so on. The $class_{n-1}$ will have $(N-3) \times S$ empty slots in its table. The total number of empty spaces will be:

$$\text{total empty space} = \frac{(N-3)(N-4)}{2} \times S$$

Since $S = \frac{M}{N-2}$

$$\text{total empty space} = \frac{(N-3)(N-4)}{2(N-2)} \times M$$

Thus, the total empty spaces X are bound by:

$$X \leq \frac{(N-3)(N-4)}{2(N-2)} \times M$$

The ratio of total empty spaces and used spaces, R, is:

$$R \leq \frac{1}{4} \cdot \frac{(N-3)(N-4)}{(N-2)} \tag{4.2}$$

For the *CIT* technique, since all the vertexes in the graph have the same degree $\frac{M}{(N-2)}$, The worst result is which all of $class_2$ to $class_N$ have the maximum size M. Thus the ratio of empty space of **CIT**, R is:

$$R = \frac{1}{2} \cdot (N-3) \tag{4.3}$$

And the ratio of **CCIT** and **CIT** is:

$$\text{CCIT} : \text{CIT} = \frac{1}{2} \tag{4.4}$$

The Equation 4.4 shows, the **CCIT** technique has better performance than the **CIT** technique. On average the *CCIT* technique creates only half as many empty spaces as**CIT** technique creates.

Krogdahl was one of the first authors to address the problem of packing in the presence of multiple inheritance [Kro85]. His method requires pointer coercion and fails if an attribute is inherited along more than one path. It has the advantage, however, of not requiring global analysis at link time. Stroustrup's proposal for C++ [Str87] is an extension of Krogdahl's method which overcomes the multiple path problem but introduces complicated coercions and some space overhead. The current approach used by C++ [ES90] is essentially the same and appears to be a good solution given the constraint of a standard linker.

Our method is quite similar to that of Dixon *et al.* [DMS89]. The significant difference is that we split the inheritance graph into subgraphs before beginning to colour it. We have found experimentally that this improves the packing densities.

The approach used by Connor *et al.* [CDMB89] is slightly different. In their model, an object reference contains a pointer to an address map and another pointer to the actual data fields. This technique is not as efficient as ours, but its greater flexibility makes it more suitable for systems in which classes can be created at run-time.

Pugh and Weddel [PW90] show that dense packing can be achieved if records are allowed to grow in both directions. In other words, slots may have both positive and negative offsets with respect to a pointer to the object or class. Their method may achieve better packing density at the expense of considerable higher computational complexity.

Chapter 5

Conclusion

In this thesis, we have discussed the implementation of Dee. Dee is an object oriented programming language with an interactive programming environment. The goal of this research is to implement a object oriented language and its programming environment at the same time in order to achieve a better integrated system organization and performance.

The following is a summary of the research and contribution of this thesis.

5.1 CIM

In Dee, the programming environment is an integral part of the language. We developed both the language and the environment at the same time. The class interface manager is the fundamental counterpart of our system. The advantages of this design and implementation are:

1. **Centralizated control**

   As Figure 1.1 shows, all major system components communicate with the CIM by using the class interface. The role of the CIM is like an interface server, responsible for data storage and information retrieval. The compiler passes the validated class interface to the CIM and also sends queries to the CIM. All user source documents are private to themselves, but the class interfaces would be a public properties of a group of users. This accessing scheme gives the development environment a very clean and easy controlled facilities to share coding and the information. By doing this, data consistency and integrity are guaranteed.

2. **Validated information**

   In Dee, the only task updating the data base is the compiler. All data passed to CIM is validated and accepted by the compiler's semantic checking. CIM guarantee that all information provided to users is up to date and valid. Both the system and the users can rely on the information they get from the CIM.

3. **Tightly coupled system**

   The major principle of the design of Dee is to provide a tightly coupled interface between other components. The CIM provides different *views* to different components of the system. The semantic checker of the language looks to the CIM like a class interface server, providing class interface in the form of an AST. The linker of Dee sees the CIM as an information retriever, providing class information in the form of attribute list. The semantic browser and the graphical user interface see the CIM as a information display board, which can retrieve and redirect the output information according to thier needs.

For overall consideration, CIM is not just a database management unit, it also simplify the name symbol maintenance in the compiler, narrows the interface between each components. and makes each component concentrate only on its own functional aspect.

4. **No need for extra compiling**

   Views of CIM are in the form of the data structures which can be directly used by the retriever. No extra compiling effects need to take place in the semantic checker or linker. Other languages like Eiffel and C++ need to re-compile the class interface header in order to perform the semantic checking and the object linking.

5. **Project Management and System Information**

   The CIM maintains precise control over the content of the data base and *views*. In particular, it can provide a distributed development environment by maintaining the original source code as private files and the database as a shared resource.

   The current implementation of CIM not only stores the class interface. But it is also responsible for storing some extra information about the development system, such as locations of files, modification times, interface changing times, etc,.

6. **The Extendibility of CIDB**

   As an ongoing development project, the major design issues of CIM and CIDB are to provide a extendable, easy accessible and maintainable system. As the discussion of Chapter 1, both CIM and CIDB had achieved high quality on these issues. Specially, the logical variable record layout of the CIDB gives a lot of flexibility for further development. The internal structure of the CIM has been classified into four layers, each layer having its own unique functionality. This makes the whole system very easy to enhance and to modify.

5.2 The environment

One significant difference between Dee and other languages is that our our language and its programming environment were developed at the same time.

The Dee system consists of an object oriented language, semantic browser, graphical class hierarchy browser and an *emacs* Dee model editor. It offers all the basic tools of software development: modifying and creating components, finding and understanding components.

As a result of its interactive programming environment, Dee provides an easy to learn and easy to use interface to the user. Users can perform their development in a standard terminal environment or with a graphical work station environment. Both of these two environments provide a full set of tools for development.

The Dee Folder is not only a graphical browsing tool, it also captures some basic object oriented concepts in its interface, inheritance and client relationship, and provides a better learning procedure to the users as well as encouraging users to reuse the existing classes.

5.3 Linker

The linker of Dee use a heuristic coloring method to get a better run-time space consumption than the CIT. Our CCIT technique is simple and efficient. It not only tries to color each classes by its inheritance hierarchy, but also tries to find out the densest packing layout. By doing this, most of class descriptors and objects are densely packed. In the worst case, the ratio of total empty spaces and used spaces R had been improved from CITs $R = \frac{1}{2} \cdot (N - 3)$ to $R \leq \frac{1}{4} \cdot \frac{(N-3)(N-4)}{(N-2)}$.

In our experiments, all the concrete classes in the leaves of the inheritance hierarchy are densely packed.

Our CCIT technique yields straightforward class description and efficienct dispatching.

5.4 Further Work

For the Dee project, we have already built up the basic components of our system. A working version is installed on the SUNKISD network, and there some students trying to learn and use it for their course project and assignments.

Since the Dee project is still in its infancy, a lot of research can be done upon the current implementation. The following is a list for long term projects that need to be done in the future.

1. **The CIM**

   The current implementation of CIM is a multiple user version. For a distributed object oriented development environment, the CIM should have been upgraded to network server version. In this version it should provide centralized data storage control and the facilities to allow user share the interface and also the object code "C" of the classes.

2. **The environment**

   There are both long term projects and short term projects that need to be done in order to enhance the Dee environment. The short term projects include: a class maker which can use to check the integrity of all classes and bring all the classes into an up-to-data state; an interactivate tutorial facility should be added to the Dee Folder. This tutorial should include lessons for the concepts of object oriented paradigm and the Dee language.

To reach the final milestone of Dee, an object oriented development environment, the long term projects should include: A version and project controller, to control and maintain the versions or software development projects; A specification language and a object oriented design methodology should be incorporated into Dee; and for the documentation purposes Dee should have self documentation capabilities to help user to create standard documentation of the classes.

For rapid prototyping development, it is neccessary to have a interpreter development environment in which the development cycle would be more flexible to the user and more suitable for software prototyping and specification.

3. **The Linker and the code generator**

The current implementation of the linker basically addresses the problems of dynamic binding and message dispatching. In order to achieve more run-time efficiency, the Dee should provide some *dynamic* to *static* binding analysis which can find out all the static binding messages and eliminate the unnecessary dynamic message dispatch at run-time.

In this case both the code generator and linker need to be enhanced.

4. **The Dee Language**

For the language itself, the further research should focus on the issue of *constant objects*, *delegation* and *persistent objects*. The issue of constant objects and delegation mechanism can enhance the flexiblity of the language. The difficulty is to avoid import and export clauses, and the type confirmance rules of delegation. The persistence issues is the fundamental topic of object oriented data bases, and is a very important enhancement to the Dee language to support data base maintenance facilities.

5. **Class Library**

The issues of object oriented paradigm is try to address the problems of code reusablity and extendability. Besides the language, the standard of class libraries is also very important. The further developmert should focus on the standardization of class libraries and develop libraries for general applications. For examples, data structure library, graphical interface library etc,.

Appendix A

CIM Interface

The following is a list of the interface of the CIM of Dee. They are listed in two separated categories. The first one is for the compiler. The second one is used by linker, browser and the Dee Folder.

INTERFACE FOR THE COMPILER

- **int CIM_Init()**

  Function initializes the CIM and CIDB.

- **int CIM_Write_Class(AST ClassNode)**

  Function write the give class interface to the CIDB. This update will be performed on the users own image.

- **AST CIM_Read_Class(char *ClassName)**

  Function return the class interface of a specified class.

- **int CIM_Get_Class_Params(char *ClassName, AST *result)**

  Function return a AST contains the class parameter information only.

- **int CIM_Get_Ancestor_List(char *ClassName, AST *result)**

  Function return a AST list which contains the ancestors of the specified class.

- **int CIM_Get_Attribute_Item(char *ClassName, char *AttrName, AST *result)**

  Function return the AST which contain the signature of a specified attribute of the given class.

- **int CIM_Close()**

  Function terminates the CIM data base system.

- **int CIM_Is_RootClass(char *ClassName)**

  Query: Is this a root class?

- **char *CIM_Get_ClassInfo(char *namePtr, IOBUFFER info)**

  query: return all the information of the given class. The information include: inheritance, ancestors, uses and location.

- **char *CIM_Get_ClassLocation(char *ClassName)**

  query: get the location of the given class.

- **int CIM_ClassHasSpecial(char *ClassName)**

  query: Does this class have special methods?

- **SAL CIM_GetSALAttributes(char *ClassName)**

  query: Get all the information of attributes of the given class. The output information include: type, implemented by, from where.

- **cim_ClassIOList CIM_Get_AllClasses()**

  query: Get all the name of classes which are used in CIM.

- **int CIM_CheckClassExist(char *ClassName)**

  query: Does this class exist in the CIDB.

- **void CIM_ReadCDisplay(char *ClassName, long switch)**

  query: Display all the information which specified in *switch* of the given class. CIM will redirect all the output to the standard output device.

- **void CIM_Read_CAPDisplay(char *ClassName, char *AttrKey,sw)**

  query: Find all the attribute which have the same name prefix as the given attribute key of the given class. Then redirect the output data to the standard output. The given switch is used to control the detail level of the information which should be display.

- **void CIM_Read_XDisplay(char *ClassName, long switch)**

  query: Display all the information which specified in *switch* of the given class. CIM will redirect all the output to a graphic window.

- **void CIM_Read_XAPDisplay(char *ClassName, char *AttrKey,sw)**

  query: Find all the attribute which have the same name prefix as the given attribute key of the given class. Then redirect the output data to a graphic window. The given switch is used to control the detail level of the information which should be display.

Appendix B

Example Programs

B.1 Class Filter

```
class Filter
-- A component of Eratosthenes' sieve
var p:Int -- Our own prime number
var f:Filter -- A filter for numbers we cannot process
public cons make (prime:Int)
-- Construct a filter for the prime
    begin
    p := prime
    f := nil
    (p.show + " ").print
    end
public method process (n:Int)
-- Ignore multiples of p but pass on non-multiples to the next filter
    begin
    if n mod p != 0
    then
        if undefined f
        then f.make(n)
        else f.process(n)
        fi
    fi
    end
```

B.2 Class Program

```
class Program
-- This class provides a keyboard for input and a screen for output. A
-- descendant class should provide a method entry which begins by calling
-- self.open and ends by calling self.close.
public var in: Stdin
-- Use stdin for input.
public var out: Stdout
-- Use stdout for output.
method open
-- Construct and open the keyboard and window.
    begin
    in.assign
    out.assign
    end
method close
-- Close the keyboard and window.
    begin
    in.close
    out.close
    end
method reply (message:String): String
-- Display the argument and return the user's reply.
    begin
    out.write(message)
    result := in.getline
    end
```

B.3 Class Primes

```
class Primes
-- Compute prime numbers using Eratosthenes' sieve
inherits Program
method entry
-- This method is the entry point of the executable class.
var f:Filter i:Int max:Int
    begin
    self.open
    max := 100
    f.make(2)
    attempt
        from i := 3
        until i ¿ max
        do
            f.process(i)
            i := i + 1
        od
    handle e:Int "Failed".print
    end
    self.close
    end
```

Bibliography

[Bar84] Richard M. Barstow. Emacs: The extensible, customizable, self-documenting display editor. In David R. Barstow, editor, *Interactive Programming Environments*, pages 300–325. McGRAW-HILL Book Company, 1984.

[BDMN73] G. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA Begin*. Petrocelli/Charter, 1973.

[BR89] T. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. In T. Biggerstaff and A. Perlis, editors, *Software Reusability. Volume I: Concepts and Models*, pages 1–17. ACM Press (Addison Wesley), 1989.

[Bud91] T. Budd. *An Introduction to Object-Oriented Programming*. Addison Wesley, 1991.

[CDMB89] R. Connor, A. Dearle, R. Morrison, and A. Brown. An object addressing mechanism for statically typed languages with multiple inheritance. In N. Meyrowitz, editor, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 279–286, 1989.

[CG92] B. Cheung and P. Grogono. Compact record layouts for multiple inheritance. In *European Conference on Object Oriented Programming*, 1992. Submitted.

[DMS89] R. Dixon, T. McKee, and P. Schweizer. A fast method dispatcher for compiled languages with multiple inheritance. In N. Meyrowitz, editor, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 211--214, 1989.

[ES90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[Fat88] Richard Fateman. *Common LISP: The reference.* Addison-Wesley, 1988.

[GC91] P. Grogono and B. Cheung. Database support for browsing. Technical Report OOP–91–1, Department of Computer Science, Concordia University, January 1991.

[Gol84] A. Goldberg. The influence of an object-oriented language on the programming environment. In D. Barstow, H. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, chapter 8, pages 141–174. McGraw-Hill, 1984.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Gro90] P. Grogono. The book of Dee. Technical Report OOP–90–3, Department of Computer Science, Concordia University, February 1990.

[Gro91a] P. Grogono. The dee report. Technical Report OOP-91-2, Department of Computer Science, Concordia University, January 1991.

[Gro91b] P. Grogono. Issues in the design of an object oriented programming language. *Structured Programming*, 12(1):1–15, January 1991.

[Heg92] Lawrence A. Hegarty. *Implementing the Dee System: Issues and Experiences.* Master thesis,Concordia University, 1992.

[INC88] Franz INC. *Allegro Common Lisp User Guide: Release 3.0.* Franz Inc, 1988.

[INC91] Franz INC. *Allegro Common Lisp User Guide: Release 4.0.* Franz Inc, 1991.

[Kee89] Sonya E. Keene. *Object Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley, 1989.

[KL89] W. Kim and F. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications.* ACM Press (Addison-Wesley), 1989.

[KM90] T. Korson and J. McGregor. Understanding object oriented: a unifying paradigm. *Comm. ACM*, 33(9):41–60, September 1990.

[Kro85] S. Krogdahl. Multiple inheritance in Simula-like languages. *BIT*, 25:318–326, 1985.

[LO90] Y. Li and T. O'Shea. BRRR: a tool for facilitating user's navigation in Smalltalk-80. In *Proc. Symp. on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 175–189, September 1990.

[McC81] J. McCarthy. History of LISP. In R. Wexelblat, editor, *History of Programming Languages*, pages 173–196. Academic Press, 1981.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Prentice Hall International, 1988.

[Mey90a] B. Meyer. Lessons from the design of the Eiffel libraries. *Comm. ACM*, 33(9):68–88, September 1990.

[Mey90b] S. Meyers. Working with object-oriented programs: the view from the trenches is not always pretty. In *Proc. Symp. on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 51–65, September 1990.

[ND81] K. Nygaard and O-J. Dahl. The development of the SIMULA language. In R. Wexelblat, editor, *History of Programming Languages*, chapter IX, pages 439–493. Academic Press, 1981.

[Nie89] O. Nierstrasz. A survey of object oriented concepts. In W. Kim and F. Lochovsky, editors, *Object Oriented Concepts, Databases, and Applications*, pages 3–21. ACM Press (Addison Wesley), 1989.

[PW90] W. Pugh and G. Weddell. Two-directional record layout for multiple inheritance. In *ACM Conf. on Programming Language Design and Implementation*, pages 85–91, 1990.

[Sak88] M. Sakkinen. On the darker side of C++. In S. Gjessing and K. Nygaard, editors, *Proceedings of the 1988 European Conference of Object Oriented Programming*, pages 162–176. Springer, 1988. LNCS 322.

[SB86] Mark Stefik and D.G Bobrow. Object-oriented programming: Themes and variations. *The AI Magazine*, 6(4):182–204, 1986.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Str87] B. Stroustrup. Multiple inheritance in c++. In *Proceedings of the European Unix Users Group Conference*, pages 189–207, May 1987.