

Issues in the Design of an Object Oriented Programming Language

Peter Grogono

Department of Computer Science, Concordia University
1455 deMaisonneuve Blvd. West, Montréal, Québec
Canada H3G 1M8

Tel: (514) 848-3000

Fax: (514) 848-2830

e-mail: grogono@concour.cs.concordia.ca

Published in *Structured Programming*, January 1991.

Abstract

The object oriented paradigm, which advocates bottom-up program development, appears at first sight to run counter to the classical, top-down approach of structured programming. The deep requirement of structured programming, however, is that programming should be based on well-defined abstractions with clear meaning rather than on incidental characteristics of computing machinery. This requirement can be met by object oriented programming and, in fact, object oriented programs may have better structure than programs obtained by functional decomposition.

The definitions of the basic components of object oriented programming, object, class, and inheritance, are still sufficiently fluid to provide many choices for the designer of an object oriented language. Full support of objects in a typed language requires a number of features, including classes, inheritance, genericity, renaming, and redefinition. Although each of these features is simple in itself, interactions between features can become complex. For example, renaming and redefinition may interact in unexpected ways. In this paper, we show that appropriate design choices can yield a language that fulfills the promise of object oriented programming without sacrificing the requirements of structured programming.

1 Introduction

The goal of structured programming is to find abstractions that enable us to build correct and efficient computer programs. The idea that an object is a useful abstraction emerged in the design of SIMULA 67 [1], inspired Smalltalk [2], and recently has achieved widespread popularity. Despite its antiquity, the notion of an object is still fluid and the subject of vigorous debate.

Structured programming is strictly unnecessary because we could, in principle, write all our programs in machine language. Thus the goal of structured programming cannot be expressed in precise terms such as correctness or completeness. Nevertheless, the central objective of structured programming is clear: it is to find *abstractions* that express computations. The most useful abstractions yield programs which are easy to read, easy to reason about, and efficient to execute.

We can trace the evolution of object oriented programming by considering the roles of scope and extent in the history of programming languages. The physical distance between the definition and use of an identifier has a significant effect on the readability of a program. The *scope* of an identifier is the region of program text in which the identifier may legitimately be used. The *extent* of a variable is the period of time for which it exists during execution. We can classify programming languages in terms of their treatment of scope and extent. In the earliest languages, all variables had global scope and infinite extent. FORTRAN introduced limited scopes. Algol introduced limited extent, described by nested blocks and implemented by stack allocation. Although stack allocation had the important consequence of supporting recursion, the close coupling of scope and extent was occasionally too restrictive. The **own** variable of Algol 60 was an early but ultimately unsuccessful attempt to overcome the restrictions, but it led to the concepts of *class* and *object* of SIMULA 67.

A *class* defines a scope within which the attributes of an object are declared. Some of these attributes are visible outside the class and provide the means of manipulating the object. A class provides a means of instantiating new instances of the class which are called *objects*.

Although the definitions of “class” and “module” differ between programming languages, we can say broadly that a class is a kind of module. The difference between modules and classes lies in the nature of the entities which may be exported. In most modular languages, a module may export arbitrary entities, including types, constants, variables, and procedures. In most object oriented languages, a class can export only procedures which perform operations on instances of the class. The difference leads directly to one of the major issues of structured programming. Is the greater generality of modules necessary? Do the restrictions on classes unduly limit the expressiveness of the language?

Two other research topics of the seventies contributed to the development of classes and objects. *Information hiding* [3] demonstrates the importance of the module interface as a contract between client and supplier. The development of *abstract data types* [4] demonstrates the importance and usefulness of separating specification and implementation.

The philosopher’s stone of programming is *reusable code*. All programmers have experienced *déjà vu* as they write yet another slight variation on a table search. The key to reuse is abstraction and parametrization [5]. The object oriented paradigm provides both. Most object oriented languages support inheritance, a mechanism which enables the common features of

a group of classes to be abstracted into a parent class. Some object oriented languages also support parameterized classes, which enable one class to do the work of many.

In this paper, we describe the design of a simple object oriented language [6]. The language is called “Dee”, after the British mathematician and philosopher John Dee, 1527–1608. Dee is a strongly typed, object oriented language with facilities for developing and maintaining programs [7]. In fact, the primary objective in the design of Dee was that programs written in it should be easy to maintain. A second objective was to keep the language simple.

In Section 2, we discuss the objectives of the language design exercise and the criteria we imposed on the new language. In Section 3, we give a brief description of Dee, highlighting its unusual features. In Section 4, the major section of the paper, we describe some of the problems which arose during the design of Dee and our solutions to them. In Section 5, we discuss some aspects of object oriented programming style which go beyond most discussions of this topic in recent publications. Finally, in Section 6, we discuss the lessons we have learned from this exercise in language design.

2 Objectives and Criteria

A useful program may have a lifespan of many years. During that time, it will be maintained, probably by many different programmers. Three kinds of maintenance are recognized in software engineering: perfective, adaptive, and corrective [8, Chapter 27]. *Perfective* maintenance improves the program without changing its functionality. *Adaptive* maintenance responds to changing requirements and compensates for changes in the environment in which the program is used. *Corrective* maintenance diagnoses and rectifies previously undiscovered errors.

The most difficult task confronting a maintenance programmer is to understand existing code. In many cases, the code was written by a programmer who is no longer available for consultation. The programming language and the tools that support it must be designed to assist the maintenance programmer as much as possible.

Consideration of the primary objective of maintainability led to six design principles for Dee. The first of these principles was that *information should not be duplicated*. Redundancy may be useful, but duplication is not. The second principle was that *all information related to a program entity should be in one place*. Both of these principles contribute directly to maintainability. They have several immediate consequences: for instance, Dee programmers do not write separate files for specifications and implementations; there is no need for separate documentation files; and there are no import or export lists in class declarations.

The third principle is that *the programmer should not have to provide information that the compiler can easily infer and display*. For example, Dee programmers do not have to write the names of attributes inherited from another class. The compiler can infer the names and make them available when they are needed by programmers. Conversely, programmers should provide information that the compiler cannot easily infer, such as whether a method is intended to be used as a constructor.

The fourth principle is that *the semantic analysis performed by the compiler should not be complicated*. For example, an elaborate system of automatic conversions requires the compiler to

analyse many different cases. This principle may seem surprising at first sight. The tendency has been for compilers to become more complex as languages have become more advanced.

The meaning of a program should be evident from its text. If it is difficult for a compiler to understand a program, it will be difficult for people also. We use the phrase “semantic analysis” in the strict sense of checking that the program is semantically correct. The compiler of an object oriented language may need to perform extensive analysis to obtain efficient code, but we consider this to be part of optimization, code generation, or linking, rather than of semantic analysis.

The fifth principle is that *a language which is intended for the development of production quality software should be strongly typed*. A program with type declarations is easier to read and modify than an untyped program. We do not require that a type-correct program should never raise an exception at run-time, but we do require that it should never fail in an unanticipated way.

Strong typing is motivated by the principle that semantic analysis should be simple, but there are other reasons for desiring it. While it is true that, if a language is suitably designed, the compiler can infer types in the absence of type declarations [9, 10], and that type inference is convenient for interactive and prototyping languages [11], untyped programs are not suitable for production software. The benefit from automatic type inference is that programs are slightly shorter. The loss is that programs are harder to understand and compile. With a little extra effort from the programmer, the tasks of both the compiler and the maintenance programmers can be simplified.

The sixth and final principle is that *a language should support the chosen programming paradigm completely and consistently*. For example, in some languages, the assignment statement $x := y$ may have value or reference semantics depending on the types of x and y . This kind of inconsistency draws the attention of programmers to the underlying implementation and weakens the coherence of their conceptual model of the language. Many programmers are reluctant to give up inconsistencies to which they have become accustomed because they feel a need for “flexibility.”

Programmers want flexibility and low-level features for two reasons. The first is that they need to write low-level, machine-oriented programs. This requirement can be accommodated by a relatively small number of carefully designed features, as Wirth demonstrated in the design of Modula-2 [12]. The second reason is that programmers want their programs to be “efficient.” Almost any program will be efficient if it meets three conditions: (1) the compiler implements the basic operations of the language efficiently; (2) the programmer chooses appropriate algorithms; and (3) the programmer ensures that the most frequently executed code segments (usually the inner loops) are identified and carefully coded. If these conditions are satisfied, the additional improvement that can be obtained by writing tricky code is usually small and is often outweighed by the extra work of debugging and maintaining the program [13].

The ideal programming language would be both simple and efficient. There are situations in which the desire for simplicity and efficiency work together. Some of the most interesting issues in language design arise, however, when it is necessary to establish a balance between simplicity and efficiency. As an example, consider the representation of objects in an object oriented language. There is a choice between storing the object itself or a reference to it. The language designer’s choices include the following.

- Represent all objects by references, as in LISP and its descendants. This strategy is simple to implement but potentially inefficient.
- Represent objects of a few basic classes by values and all others by references, as in early versions of Eiffel [14]. This strategy is likely to lead to semantic inconsistencies of the kind already described.
- Give the programmer full control over the representation, as in C++ [15]. This strategy complicates both the language definition and the programmer's task.
- Let the compiler choose the appropriate representation. This strategy is likely to complicate the compiler, perhaps to the point of infeasibility.

In cases such as these, our goal was to keep the syntax and semantics of the language simple and to be faithful to the object oriented paradigm. We accepted that the implementation, at least in early versions, would have inefficiencies, but it is usually easier to improve an implementation than it is to improve a language design.

3 Realizing the Objectives

In this section, we review some of the novel aspects of Dee. The design of Dee was based on our experience of object oriented programming, which came primarily from SIMULA 67, Smalltalk, and Eiffel. Syntactically, Dee resembles Pascal and Eiffel.

A Dee program is a collection of *classes*. There are three relations between classes: *uses*, *extends*, and *inherits*. The *uses* relation may have cycles. For instance, the class `String` uses the class `Bool` for string comparison and the class `Bool` uses the class `String` to convert Boolean objects to the strings "true" and "false."

A *client* class uses another class. A *descendant* class extends or inherits another class. The relations *supplier* and *ancestor* are the inverses of client and descendant, respectively. The relation *needs* is defined to be the reflexive and transitive closure of the union of *uses*, *inherits*, and *extends*. A *program* is defined by its *root class*: it consists of the root class and all the classes needed by the root class. Thus a program is a rooted, directed, cyclic graph.

We assume that each class will be developed and maintained by a programmer or team of programmers, therefore we define the *owner* of the class to be this programmer or team. The owner is responsible for a single document called the *canonical document* of the class. No person other than the owner of the class should need to see the canonical document in its entirety.

Within the canonical document, there are three levels of text [16]. The criterion that separates the levels is the extent to which the text can be processed automatically. The lowest level is source text, which can be compiled into machine code and can therefore be considered to be fully processed. The highest level is natural language commentary, which can be processed only in very limited ways. For example, the compiler can generate and concatenate comments. Between these extremes, there is an intermediate level at which text can be partially processed. The intermediate level of text consists of statements of a logical theory. Intermediate text has a function similar to the assertions of Eiffel. Since the notation is formal, the compiler can

parse these statements and can manipulate them in simple ways. For example, it can construct conjunctions and implications and apply syntactic transformations to simplify the resulting statements.

Each level of text has a well-defined purpose: the comments help programmers to understand the class definition; the formal statements facilitate reasoning about the class; and the source text enables the compiler to generate machine code. Since the intermediate and source text are both formal, it may be possible in principle to prove that the source meets the specifications, but the compiler does not attempt to construct such proofs. This is another reason for saying that the intermediate level of text is only partially processed.

Each of the three levels of text is further divided into *private information*, visible only to the owners of the class, and *public information*, visible to owners of other classes through views, which we describe below.

Programmers need rapid access to accurate and up-to-date information about the programs they are developing. If the languages they use provide classes and inheritance, they need to know what classes can do and how they are related to one another. The class hierarchy browser of Smalltalk provides some of this information; becoming familiar with it is an important part of learning Smalltalk. The **short** and **flatten** utilities of Eiffel extract information from source code. In Dee, the information is provided by views.

A *view* is a human-readable text intended for a programmer who wants to use, extend, or inherit a class which he or she does not own. This definition is consistent with Shilling and Sweeney, who define a view to be “a simplifying abstraction of a complex structure” [17]. The *client view* contains information of the kind that would be found in a specification module in Modula-2. The *descendant view* contains additional information, reflecting the fact that the descendant relation is more intimate than the client relation. In each case, the view contains selected documentation.

An *interface* is a machine-readable file that is read by the compiler when it is compiling another class. An interface contains only the information that is needed by the compiler. An interface may contain information that is not needed by and should not be used by the programmer. This information may include, for example, object sizes, variable offsets, or invocation protocols. When it compiles a class, the compiler generates views and interfaces for the class.

Views and interfaces are always up to date because they are generated by the compiler. We have seen that the structure of a Dee program is in general a cyclic graph with a class at each vertex. The number of classes is quite large: even a small program may require a hundred or more classes. Consequently, programmers make frequent use of views. A programmer can access a view quickly from the editor by moving the cursor onto a class name and pressing a single key.

Figure 1 shows the canonical document of a simple class, called Example. The class Example inherits a class Parent, has two instance variables, *n* and *x*, and has a constructor *make*. Dee classes typically contain more comments and white space than are shown in Example. This class, like the others presented in this paper, has been stripped to the bare minimum necessary for explanation.

Figure 3 shows the client view of the class Example and Figure 2 shows its descendant view. Both views include selected documentation. The public attribute *n* is included in both the client and descendant views. The attribute *x* is private because the programmer has not specified

```
class Example
inherits Parent
public var n:Int
var x:Float { private by default }
public cons make (xi:Float)
  { A constructor for instances of Example }
begin
  n := 0
  x := xi
end
```

Figure 1: The class Example

```
class Example
inherits Parent
ancestors GrandParent Parent
n:Int
x:Float { private by default }
proc move { Inherited from Parent }
cons make (xi:Float)
  { A constructor for instances of Example }
```

Figure 2: The descendant view of class Example

```
class Example
inherits Parent
ancestors GrandParent Parent
n:Int
proc move { Inherited from Parent }
cons make (xi:Float)
  { A constructor for instances of Example }
```

Figure 3: The client view of class Example

public. It is therefore included in the descendant view but not in the client view. Both views include selected comments. The compiler has added the relation ancestors, the transitive closure of inherits.

```
class Example
ancestors GrandParent Parent
public var n:Int
public proc move
public cons make (xi:Float)
```

Figure 4: The client interface of class Example

Figure 4 shows the client interface of class Example. The descendant interface is similar but includes inherited information.

In the current version of Dee, the views and interfaces of a class are stored in text files. In the next version, all of the information will be stored in a central database. The purpose of this projected change is twofold: it should both reduce compilation time and provide a foundation for improved browsing facilities.

Syntactically, a class consists of a name, a list of class parameters, a list of inherited and extended classes, and a list of attributes. Each attribute has several properties which are, as far as possible, independent of one another.

- An attribute is either a *variable* or a *method*. A method is a *function*, a *procedure*, or a *constructor*.
- An attribute may be *public* or *private*. Only public attributes are visible to client classes.
- An attribute may return a value.
- Evaluation of an attribute may alter the state of the current object or one of its components, in which case we say that it has *side-effects* or that it is a *mutator*. An attribute which does not have side-effects is called *pure*.
- An attribute may be implemented in the class in which it is first defined or in a descendant of that class. An attribute is called *abstract* in a class in which it is defined but not implemented and *concrete* in a class in which it is implemented.
- Each attribute has a *signature* consisting of its name, the classes of its arguments (if any), and the class of the value that it yields (if any). An attribute may inherit its signature, or its signature and implementation, from an ancestor.

Variables and functions are pure, but procedures and constructors may have side-effects. Variables, functions, and procedures may return values, but constructors do not return values. The distinction between functions and procedures in Dee is different from that in many other languages: it refers to side-effects rather than to the ability to return a result.

In the canonical document, the owner of the class declares a variable with the keyword `var` and a method with one of the keywords `method` or `cons`. Figure 5 compares the declarations

	Variable	Function	Procedure	Constructor
Source	var	method	method	cons
View	-	-	proc	cons
Interface	var	func	proc	cons

Figure 5: Attribute Declarations

written by the owner in the canonical document of a class (called “source” in Figure 5) to the annotations written by the compiler in the views and interfaces of the class. A dash indicates that no symbol is written. The programmer of a client class cannot distinguish between a variable and a parameterless function. On the other hand, programmers of client and descendant classes can tell whether a method has side-effects although this information was not provided by the owner of the class. The distinction between variables and functions is passed to the client interface so that the compiler can generate appropriate code. This is one example of the advantage of separating human and machine-readable interfaces.

Calls in Dee have the form $x.m(a)$, in which x is either the name of an object or an expression which yields an object, m is the name of a method, and a is an argument. This particular call has exactly one argument, but in general there may be zero or more arguments. We refer to x as the *first* argument or *receiver* of the call and to a as the *second* argument. If x has a public instance variable, v , the expression $x.v$ yields its value.

4 Design Issues

Although the object oriented paradigm is easy to explain and understand, a number of interesting issues arise in the design of an object oriented language. Language design decisions reflect the overall goals of the designer. Smalltalk offers flexibility and a rich, interactive, development environment; C++ is similar to C and is efficient; and Eiffel supports the development of large programs. As we have seen, the major design goals of Dee were simplicity and maintainability. These goals interact with semantic aspects of the language in various ways. In this section, we discuss the effect of the goals on the design of the language.

4.1 Names and Objects

An identifier in a program may denote either a reference to an object (*reference semantics*) or the object itself (*value semantics*). Dee has a reference semantics for assignment and argument transmission. The assignment $x := y$ makes x refer to the same object as y : it never creates a new object. Reference semantics are not appropriate for comparison of objects, however. Section 4.5 describes the methods that Dee provides for comparing objects.

The syntax of Dee permits only simple names on the left of the assignment operator. An assignment of the form $x.a := y$ is not allowed because it would alter an attribute of the object x . The only way to alter an object is to invoke one of the methods of its class.

A concrete class must have constructors, but it need not have mutators. A class without mutators is called a *pure class* and its instances are called *immutable objects*. The basic classes

Int, Float, Bool, and String are pure classes. A program which uses pure classes is, for all intents and purposes, a functional program.

4.2 Classes and Types

Dee is strongly typed and follows the convention that each class defines a type. Since inheritance and subtyping conflict [18], Dee provides two kinds of subclass, described in Section 4.3 below. Inheritance provides two of the features we need in order to build reusable software components: polymorphism and overloading. Class parameters are not required in principle but they enhance the expressiveness of the language [19].

In Dee, a class parameter must be qualified. For example, the declaration

```
class Table [KeyType:Ordered; InfoType:Any]
```

introduces a class Table. The class Table is parameterized by a class of keys, which must be ordered, and by a class of objects of arbitrary type. A class which is a client of Table might declare

```
phonebook: Table[String Person]
```

The class Any exists to provide a sensible default for parameter qualification. It is not an ancestor of every class and it has only a small number of attributes, unlike the class ANY of Eiffel. The kind of polymorphism exhibited by Table, in which class parameters are restricted, is called “bounded parameteric polymorphism” [20] or “constrained genericity” [14].

The concepts *class* and *type* in Dee are closely related. For basic classes, they are essentially identical. It does not matter whether we read `n:Int` as “`n` belongs to class Int” or “`n` has type Int”. For generic classes, however, we make a distinction: we say that Table is a class and that `Table[Int String]` is a type.

A type *X* conforms to a type *Y* if an instance of *X* can be used in any context in which an instance of *Y* is acceptable. Intuitively, for example, Apple conforms to Fruit because statements about fruits are true of apples. For simple types, the rules of conformance are straightforward. A class *X* conforms to its own type *X*. If *X* is a descendant of *Y*, then *X* conforms to *Y*.

Conformance rules determine the validity of signatures in descendant classes. Here is a partial declaration of the class Fruit.

```
class Fruit
  proc make (x:Fruit)
  func this:Fruit
```

Suppose that class Apple inherits class Fruit. The methods `make` and `this` must both be exported by Apple. If they are not redefined in Apple, their signatures will be changed by the compiler. The user’s view of Apple will be:

```

class Apple
inherits Fruit
proc make (x:Fruit)
func this:Apple

```

The type of the argument x in method `make` cannot be changed to `Apple` because the method might receive an argument of type `Fruit` at run-time, as demonstrated by the following sequence of statements.

```

var a:Apple
var f,g:Fruit
.....
f := a
f.make(g)

```

The assignment `f := a` is allowed because `Fruit` is an ancestor of `Apple`. At run-time, the object corresponding to `f` is an `Apple`, and therefore the method `make` of the class `Apple` is invoked. Its argument, `g`, is a `Fruit`.

The conformance rules for parameterized classes are more complicated, as shown by Cook [18]. Consider the following simplified code for a generic class for sets.

```

class Set [T:Comparable]
proc insert (x:T)
func arb:T

```

The procedure `insert` adds a new object of type `T` to the set. The function `arb` returns an arbitrary member of the set, failing if the set is empty. We consider the types `Set[Apple]` and `Set[Fruit]` under the assumption that `Apple` conforms to `Fruit`.

If we ignore the function `arb`, then `Set[Apple]` conforms to `Set[Fruit]` because it is reasonable to insert an apple into a set of fruits. The function `arb`, however, prevents this conformance because an arbitrary member of a set of fruits is not necessarily an apple.

Conversely, if we ignore the procedure `insert`, then `Set[Fruit]` conforms to `Set[Apple]` because an arbitrary member of a set of apples must be a fruit. If we include `insert`, however, conformance fails because we cannot in general insert a fruit into a set of apples. Thus `Set[Fruit]` and `Set[Apple]` are incompatible types.

Following Cook, we define a formal type parameter such as `T` in the class `Set` to be *covariant* if it is used only as the type of an instance variable, local variable, or result and we define it as *contravariant* if it is used only as a formal parameter. A parameter which is used in both contexts is *bivariant*. In the class `Set`, `T` is covariant in `arb`, contravariant in `insert`, and bivariant in the class `Set` as a whole.

Using this terminology, we can give the rule for conformance for parameterized types. Suppose that the class `C` has the type $C[T_1, T_2, \dots, T_n]$, in which each T_i is a formal type parameter. Then $C[X_1, X_2, \dots, X_n]$ conforms to $C[Y_1, Y_2, \dots, Y_n]$ if, for each i , $1 \leq i \leq n$, one and only one of the following statements is true.

- T_i is covariant and X_i conforms to Y_i .
- T_i is contravariant and Y_i conforms to X_i .
- or T_i is bivariant and $X_i = Y_i$.

The name of an attribute is *overloaded* if it denotes more than one entity. Determining the particular entity denoted is called *resolving* the overloaded name. Most object oriented languages resolve overloaded names by examining the class of the first argument (or *receiver*) at run-time; the other arguments, if any, are not examined. This asymmetry complicates type checking [21]. It is, of course, possible to resolve overloaded names by examining the classes of all arguments of the method, as in CLOS [22], but most object oriented languages use the first argument only.

Suppose that we define an abstract class RingElem whose instances are ring elements with a binary operator $+$. Assume that RingElem has two descendants, Int and Float. Consider the following code.

```

r,s,t:RingElem
i:Int
f:Float
.....
r := i
s := f
t := r + s

```

The assignments $r := i$ and $s := f$ are allowed by the normal rules of inheritance: RingElem is an ancestor of both Int and Float. The assignment $t := r + s$ is allowed because each term is a RingElem. Yet the program as a whole is incorrect because, when executed, it will add the integer r to the float s .

The signature of $+$ in RingElem is $\text{RingElem} \times \text{RingElem} \rightarrow \text{RingElem}$. We would like the signature of $+$ in Int to be $\text{Int} \times \text{Int} \rightarrow \text{Int}$. This does not work because, although we can be sure that the first argument is an Int (this is accomplished by dynamic binding), static type checking can ensure only that the type of the second argument is RingElem. From the point of view of type theory, $\text{Int} \times \text{Int} \rightarrow \text{Int}$ is not a subtype of $\text{RingElem} \times \text{RingElem} \rightarrow \text{RingElem}$ [23].

In Eiffel, the type of the second argument of $+$ is declared to be **like current**. Meyer calls this *declaration by association* [14]. Cook has shown that declaration by association is insecure [18].

Alternatively, we can use the signature $\text{Int} \times \text{RingElem} \rightarrow \text{Int}$ for $+$ in Int and dispatch a second message with the second argument as receiver to complete the operation. This technique is called *double dispatching*. It was proposed by Ingalls for Smalltalk [24] and has subsequently been refined by Hebel and Johnson for Typed Smalltalk [25].

Double dispatching has three apparent disadvantages. First, it requires a large number of methods, many of which will be trivial. If RingElem has n subclasses, each subclass must have $n + 1$ methods just to implement $+$. Second, double dispatching will slow execution. Third, double dispatching is inconsistent with one of the goals of object oriented programming, that it should be possible to add new classes without affecting the validity of existing classes.

Hebel and Johnson show that the first problem is not as serious as one might expect from the foregoing argument. By constructing a deeper class hierarchy and exploiting inheritance, they eliminate many small methods. By code optimization, they reduce the overhead of double dispatching to an acceptable level. For a complete arithmetic hierarchy with 36 classes and 5 binary operations in each class, we would expect to need 6660 methods using double dispatching. With inheritance, only 401 methods are actually needed [25].

4.3 Inheritance

Disciplined use of inheritance is a key component of object oriented design. Like all good things, inheritance can be overused. The strict interpretation of inheritance in Dee is intended to prevent the use of inheritance as a trick to achieve conciseness and efficiency with no regard for meaning. Consider, for example, the implementation of a stack using an array in Eiffel [14, pages 241-2].

```
class FIXED_STACK[T] ...
  inherit
    STACK[T];
    ARRAY[T];
  ...
```

The required stack behaviour in the class *FIXED_STACK* is obtained by renaming array operations as stack operations. The problem with this example of inheritance is that the keyword **inherit** is used in two quite different ways. Moreover, the class *FIXED_STACK* is *logically* a client, not a descendant, of the class *ARRAY*. If we must abuse inheritance in this way for the sake of conciseness and efficiency, something is seriously wrong.

For a discussion of inheritance, we must distinguish abstract and concrete methods. A method which has an implementation in the class in which it is defined is called a *concrete method*. A method may have a signature but no implementation, in which case it is called an *abstract method*. A class is called *abstract*, *concrete*, or *partially abstract* according to whether it contains only abstract methods, only concrete methods, or a mixture of abstract and concrete methods.

Partially abstract classes, under various names, have been part of object oriented programming from the beginning. For example, Ingalls describes the class *Number* in Smalltalk-76 [26]: it implements the methods \leq , \geq , \neq , \max :, and \min :. using the abstract methods $<$, $>$, and $=$. The methods $<$, $>$, and $=$ must be provided by any concrete descendant of *Number*. It does not make sense for an abstract or partially abstract class to have instances. The rule in Dee that ensures that an abstract class cannot have instances is simple: a class may have either abstract methods or constructors; it may not have both.

The semantics of an object oriented language must assign a meaning to abstract, partially abstract, and concrete classes. In our view, an abstract class is a *theory* for which a concrete descendant class is a *model*, using the terms theory and model in approximately the sense in which they are used in mathematical logic. In this interpretation, a partially abstract class is a parameterized model.

In Smalltalk, any message can be sent to any object. If the corresponding method is not implemented by the class of the object or its ancestors, the system displays a message of the form “object O cannot understand message M”. If an object oriented language requires the class of each object to be declared, and places certain restrictions on inheritance, errors of this kind can be detected by the compiler.

If we declare `x:C`, the run-time object corresponding to `x` must belong either to the class `C` itself or to one of its descendants. It follows that, if a class is obliged to provide all of the attributes of its ancestors, the compiler can ensure that no object will ever receive a message that it cannot “understand”.

Descendant classes in Dee must export all of the public names that they inherit. They may attach a new signature and meaning to a name, provided that the signature conforms to the inherited signature in the way described in Section 4.2 above. The current version of Dee does not allow an attribute to be renamed in a descendant class because renaming may violate type security.

It is sometimes useful to exploit the features of a class without the obligation to provide all of its functionality. For this purpose, Dee provides another keyword: `extends`. The declaration “`extends P`” in a class `C` allows `C` to inherit attributes from `P` but the type `C` does not conform to the type `P`. The relations `inherits` and `extends` of Dee correspond to public and private derived classes in C++.

4.4 Object Creation and Destruction

The message `x.m(a)` is meaningful only if `x` is an existing object. A message such as `x.create`, intended to create a new object named `x`, does not make sense if `x` does not exist when this message is dispatched.

At the time when we want to create a new object, we know the class to which the object will belong. We could send a message to the class, requesting a new instance. Smalltalk objects are created in this way. If we adopt this solution, we have to accept that either object creation is an anomalous operation or that classes are themselves objects. In Smalltalk, classes are indeed objects. The Smalltalk approach has advantages, especially for a dynamic language with a rich development environment, but it also has disadvantages: it introduces the complexity of metaclasses into the language, and the object created is always in the initial state defined for the class.

In a language in which classes are not objects, we require a way of creating objects which respects encapsulation yet allows us to specify the initial characteristics of new objects. Objects of the basic classes are created by literals. For example, the literals

```
3      3.14      true      "Hello"
```

create objects of classes `Int`, `Float`, `Bool`, and `String`, respectively. The code for creating a new object of a general class in Dee has the following form:

```
x:C
.....
new x.c(a, b, ...)
```

The method `c` must have been declared as a *constructor* of the class `C`. A constructor is a method with the following properties.

- It must not make use of the values of any of the instance variables of the receiver before assigning to them, because these variables may be undefined.
- It must assign values to all of the instance variables of the receiver.
- The values assigned must establish the class invariant.

The implementation is straightforward. The effect at run-time of the keyword `new` is to allocate enough storage for an instance of the receiver class and then to invoke the constructor.

There are two apparent disadvantages with this scheme. The first is that it permits the existence of undefined objects: the object `x` above is accessible within the scope of the declaration `x:C` but is undefined until the `new` statement has been executed. This suggests that we should combine declaration and initialization in a single statement [27]. Combining declaration and initialization, however, leads to other problems [14, page 77]. First, it forces bottom-up construction of composite objects, which may not always be practical. Second, we may declare a variable in order to use it as a local copy, as in the following code.

```
x:C
.....
x := y
```

In this case, there is no point in initializing `x` at the point of declaration.

The separation of declaration and initialization does, however, complicate the implementation. Uninitialized objects must be detected either during compilation, which requires flow analysis, or during execution, which requires a representation for undefined objects.

We have mentioned that Dee uses reference semantics for assignment and argument transmission. The most straightforward way to implement reference semantics is to allocate space on the heap for every object that is not smaller than a pointer. The disadvantage of heap allocation is that heap space must be reclaimed when the object ceases to be useful. There are three ways of reclaiming storage.

First, on exit from a scope, the run-time system can reclaim the space occupied by objects declared within the scope. Second, the language can provide special forms, analogous to `new`, which allow the programmer explicitly to destroy objects. Finally, the run-time system can provide garbage collection services, reclaiming space that can no longer be accessed by the program.

We rejected the first two options and chose garbage collection for Dee. The result is that all objects have effectively infinite extent. The cost of this choice is, of course, the run-time overhead of garbage collection. This seems an acceptable price to pay for safe, simple programs and the programmer's peace of mind. Experience with languages that provide garbage collection, such as LISP, ML, and Smalltalk suggests that garbage collection increases programmers' productivity considerably.

Some objects require actions other than storage reclamation when their useful life is over. For example, when a file is closed, its buffers must be flushed and its directory entry updated. For this purpose, C++ provides *destructors*, but Dee has no explicit destructors. Instead, there is a coding convention used uniformly in the standard classes but not enforced by the compiler: a class whose objects do not require attention when they die has a constructor called *make* (and possibly other constructors). Otherwise, the class provides two methods, *open* to construct the object and *close* to destroy it.

4.5 Equality and Ordering

An object oriented language must provide ways of comparing objects for equality and, for some classes, ways of ordering objects. In most object oriented languages, including Dee, an object is either a basic value, such as an integer, or a tree with a basic value at each leaf. Suppose that x and y are references to such objects. How should we interpret the expression $x = y$?

First of all, if x and y refer to the same object, $x = y$ should be true. This kind of equality is *identity* or *reference* equality. Although a test for identity may be useful in a programming language, there must also be a way of deciding when distinct objects have equal values.

Suppose that x and y refer to different objects. In logic, x and y are *intensionally* equal if they have the same representation and *extensionally* equal if they have the same abstract value. Intensional equality implies extensional equality but the converse is not true, as the following example shows. Suppose that we have chosen to represent complex numbers using objects with three components.

```
rep:{Cartesian, Polar}
a:real
b:real
```

The object (Cartesian, x,y) represents $x + iy$ and the object (Polar, r,t) represents re^{it} . The object (Cartesian,0,1) and the object (Polar,1, $\pi/2$) both represent the complex number i . They are unequal intensionally but equal extensionally.

There are three kinds of equality which can be implemented by a compiler without assistance from the programmer. Identity is easy to implement. We can define “deep equality” as follows: x and y are “deep equal” if either they are basic objects with the same value, or they are composite objects whose corresponding components are deep equal.

Deep equality can be implemented by recursive traversal of the data structures representing the objects. If the data structures may be cyclic, the compiler must use mark bits to prevent looping.

Another kind of equality which is easily compiled is “shallow equality”. The representation of an object in an object oriented language is usually a record whose components are either simple values or pointers to other objects. The objects x and y are “shallow equal” if the records which represent them are equal, byte for byte.

Summarizing, we have three semantics for equality (identity, intensional, and extensional) and three implementations (identity, deep, and shallow). Deep equality implements intensional

equality. Shallow equality does not have any reasonable semantics, although it is occasionally what the programmer “wants”. It is clear that the compiler should not attempt to implement extensional equality. In fact, if objects are allowed to have infinite extensional values (which would be the case, for example, if functions were objects), extensional equality is not even decidable.

	x = a	x = b	x = c	x=d
Identity	T	F	F	F
Shallow	T	T	F	F
Intensional	T	T	T	F
Deep	T	T	T	F
Extensional	T	T	T	T

Figure 6: Equality Table

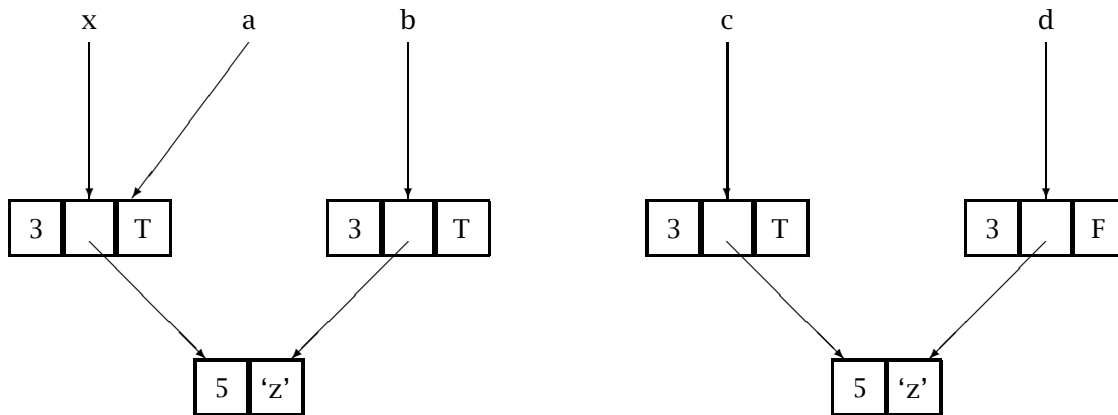


Figure 7: An Assortment of Objects

Figure 7 shows five objects, x, a, b, c, and d, each of which is represented by a record consisting of three components: an integer, a composite object containing an integer and a character, and a boolean. Figure 6 tabulates the relation between x and each of the other objects with each of the definitions of equality. We can consider identity to be the “strongest”, or most demanding, form of equality, and extensional equality to be the “weakest” form of equality. In assigning the value T to the extensional comparison $x = d$, we are assuming that the boolean component is not part of the abstract value of the object.

There are still other ramifications of equality in object oriented programming. Suppose that we have a class `LibraryBook` whose attributes include author, title, publisher, edition, year, and callnumber. It is reasonable to consider two instances of `LibraryBook` to be equal if they have the same author, title, and edition. The fact that two books have different call numbers does not make them unequal from the point of view of a reader.

Inheritance introduces further complications. Suppose the class `ValuableBook` is a descendant of `LibraryBook` with the additional attribute `loanable` (the library may want to prohibit the loaning of valuable items). If `b` is a `LibraryBook` and `v` is a `ValuableBook`, are the expressions

$b = v$ and $v = b$ legal and, if so, what do they mean? Since the method used to test for equality is taken from the class of the receiver, the expressions $b = v$ and $v = b$ may be evaluated by different methods and yield different results.

In Smalltalk, the expression $x == y$ is true if x and y are the same object. The expression $x = y$ has the same value by default, but the programmer may redefine $=$ in any class. Programmer-defined equality is not necessarily symmetric. Similarly, C++ permits the overloading of the equality operator $==$. Since the C++ processor examines the types of all arguments, however, the equality is guaranteed to be symmetric.

Dee provides three kinds of equality. If x and y belong to the same basic class (Bool, Int, Float, or String), then $x = y$ is true if x and y have the same representation. For a general class C , with instances x and y , the expression $x = y$ is defined only if the class C or one of its ancestors provides a binary predicate $=$. It is the responsibility of the programmer to ensure that $=$ implements equality in a reasonable way. Finally, the method `same` is exported by the class `Any` and may be used by descendants of `Any`; the expression `x.same(y)` is true if x and y refer to the same object.

The arguments given for equality also apply to ordering. The difference is that most languages do not provide a default ordering ($<$) predicate because the compiler cannot choose a sensible implementation without guidance from the programmer. (This is not to say that an ordering generated by the compiler would be useless, because it could be used when an arbitrary ordering is required: for example, to insert objects into a binary search tree.) Dee treats ordering in the same way as equality. The predicate $<$ is defined for basic classes but must be provided by the programmer for other classes. If $=$ and $<$ are defined for a class, the other comparison predicates may be inherited from the abstract class `Ordered`.

4.6 Collections and Iterators

A *collection* is a class whose instances are groups of objects. The classes `Array`, `List`, and `Tree` are examples of collections. All collections in Dee are parameterized by the class of the components of the collection. For example, the declaration of the class for matrices has the heading

```
class Matrix [T:RingElem]
```

which allows us to declare

```
n:Matrix [Int]
z:Matrix [Complex]
```

because `Int` and `Complex` are descendants of `RingElem` but we may not declare

```
s:Matrix [String]
```

because `String` is not a descendant of `RingElem`.

We often need to *traverse* a collection: that is, to perform an operation on each of its components in turn. The traditional structured programming approach to traversal is to exploit

the natural correspondence between data structures and control structures [28, pages 162–3]. In object oriented programming, however, we want to decouple the representation of the data structure and the control structure used to traverse it. That is, we want to write generic methods which traverse collections without knowing how the collections are represented. It follows that the *mechanism* for performing the traversal must be hidden within the class but that the *interface* to it must be shared by all collection classes. This kind of abstraction was introduced in CLU and is called an *iterator* [29].

In CLU, traversals have the following basic form:

```
for x in C.I do S end
```

In this statement, *C* is a collection, *I* denotes an iterator for *C*, and *S* is a statement sequence. The effect of the statement is to execute *S* with *x* bound to each component of the collection in turn.

In Smalltalk, traversals have a slightly different form:

```
C do: [ :x | S ]
```

The statement *S* is executed with *x* bound to each component of the collection *C* in turn. The expression [:x | S] is called a *block* and can be read as $\lambda x.S$.

These control structures are syntactically similar, although CLU offers slightly more flexibility, in that a collection can provide an arbitrary number of iterators. It is interesting to note, however, that these two control structures are in a sense inverses of each other. In CLU, the iterator yields values which are used by the caller. In Smalltalk, the caller passes a function abstraction to the collection object.

We established the following criteria for iterators in Dee. First, it should be possible to have in progress at one time more than one traversal over a particular collection. Second, iterators should not be linked to a particular control structure. The first criterion is required for operations such as closure, which examines all pairs from a collection. A consequence of this criterion is that the state of the traversal must not be part of the state of the collection. The second criterion is required because it is not always convenient to express a traversal in the form of a simple loop. A simple example is the processing of transactions against a master collection.

The effect of applying these criteria to the design is that iterators in Dee have less structure but more flexibility than the iterators of CLU and Smalltalk. An iterator in Dee is an object *I* which is associated with a collection *C*. Figures 8 and 9 show simplified declarations of the classes *Collection* and *Iterator*.

The iterator has two instance variables: the collection *col* over which it is iterating and an *ind* index to that collection. The dynamic class of the index depends on the nature of the collection. The value of the index is controlled by *init*, which sets it to the first component, and *next*, which advances it to the next component. The function *finished* returns true if there are no more components. The function *current* returns the object currently indexed. A collection must provide four functions with the same names *init*, *next*, *finished*, and *current*, but these do not change the state of the collection.

```

class Collection [T: Any]
  { Abstract class for collections }
public func init: Index
  { Returns the initial value of the cursor. }
public func finished (x: Index): Bool
  { Returns true if there are no elements for the iteration. }
public func current (x: Index): T
  { Returns the element of the collection that is currently indexed. }
public func next (x: Index): Index
  { Given the current position of the index, return the next position. }

```

Figure 8: The Abstract Class Collection

```

class Iterator [T:Any]
var col:Collection[T]
public var ind:Index
public cons make (c:Collection[T])
  begin
    col := c
    ind := c.init
  end
public proc init
  begin ind := col.init
  end
public func finished:Bool
  begin result := col.finished(ind)
  end
public func current:T
  begin result := col.current(ind)
  end
public proc next
  begin
    if not self.finished
      then ind := col.next(ind)
    fi
  end
end

```

Figure 9: The Class Iterator

```

public func sum (z:T): T
  var i:Iterator[T]
  begin
    new i.make(self)
    result := z
    from i.init until i.finished do
      result := result + i.current
      i.next
    od
  end

```

Figure 10: A Simple Application of Iterators

Figure 10 shows a simple application of an iterator. The method `sum` in this example is an attribute of the class `RingCollection`. An instance of `RingCollection` is a collection of objects each of which is an element of a ring. For example, an instance might be a linked list of integers or a matrix of polynomials. Thus `sum` is generic over both the representation of the collection and the adding operation of the ring. The only fly in this otherwise pleasing ointment is that `sum` cannot determine the zero element of the ring. If we knew that the collection could not be empty, we could select an arbitrary component `x` from it and write `x.zero` to obtain the required zero element. Since `sum` may be called with an empty collection, we must provide the zero element, `z`, to it as an argument.

It is possible to alter a collection during a traversal, for example, by adding or deleting components. Although this is usually poor programming practice, the Dee compiler does not attempt to detect it. The worst that can happen is that the user will obtain unexpected results. There is no risk of “dangling pointers” because Dee has a garbage collector.

4.7 Exceptions

As an introduction to exceptions and their handling, we consider an example which seems to require an exception handling mechanism. Suppose that we need a generic function that accepts a matrix and returns its inverse. The function is generic in the type of the components of the matrix, which may be integers, floating-point numbers, complex numbers, or polynomials. Inversion will certainly fail if the matrix is singular but it may also fail because of overflow, underflow, or exhaustion of resources. For example, inverting a matrix with polynomial components might fail for lack of memory. We assume that the inversion function is to be used in a program which inverts many matrices, only a small proportion of which are not invertible.

The first point to note is that, although inversion is a partial function, there is no point in specifying it as such because the work involved in determining whether a matrix is invertible is not significantly less than performing the inversion. Thus it is grotesquely inefficient to propose “invertible” as a precondition to be evaluated before starting the inversion.

The second point to note is that although the static context of failure is precisely known, it is not necessarily the best place at which to handle failure. Specifically, singularity will be detected by an attempt to divide by zero. We could detect exceptions by guarding each

division.

```

if y ≠ zero
  then r := x/y
  else .....

```

There are several reasons for avoiding this approach. First, it is not easy to see what the else clause should do, especially if the language does not provide non-local exits. Second, since most divisions will succeed, it is costly to execute a test before every one. Third, since the inversion function is generic, we cannot be sure that the test `y ≠ zero` is the correct criterion. For floating point matrices, for example, we need to know not that `y ≠ 0.0` but that the evaluation of `x/y` will not cause overflow. Fourth, we are not interested in which particular operation caused failure: all we need to know is that the matrix cannot be inverted. It is clear, in this example, that the exception should be handled in the dynamic scope of the failing operation rather than in the static scope.

If inverting matrices seems rather esoteric, consider a more mundane problem. The function `stoi` in Dee converts a string to an integer. The problems are similar to those of matrix inversion: `stoi` is a partial function from strings to integers, but deciding whether a given string is in its domain is hardly simpler than full evaluation. The response required after failure depends on the dynamic scope of the invocation. In Dee, for example, the idiom for reading an integer from the keyboard is the statement

```
n := kb.get.stoi
```

in which `kb` denotes the keyboard object and `get` is the method which returns a string read from the keyboard. It is quite likely that the string obtained from the user and returned by `kb.get` will not be acceptable to `stoi`. Nevertheless, we want to consider that the entire statement has failed and should be repeated.

A further observation common to these two examples is that failure should not be reported by returning a special value. The type of `stoi` is `String → Int`. If we adopt a convention such as “a result of `-99` indicates failure” we have made matters worse rather than better.

The mechanism for exception handling in Dee is based on the exception handling mechanisms of CLU [30] and ML [31]. Koenig and Stroustrup have proposed a similar mechanism for C++ [32].

The statement

```
raise x
```

signals an exception. The object `x` is called the *exception object* and may be a member of any class. Figure 11 shows the syntax of a statement containing an exception handler. The braces `{ ... }` in this figure indicate that the attempt statement may be followed by zero or more handlers.

If the class in the declaration of a handler is an ancestor of the class of the exception object, the identifier in the declaration of the handler is bound to the exception object and the statements

```

attempt
  statement-sequence
{ handle ( id : class )
  statement-sequence }
end

```

Figure 11: The Syntax of an Exception Handler

of the handler are executed. Otherwise, the run-time system tries the next handler. If no handler can process the exception, the exception propagates to the enclosing scope. If there are no suitable handlers in enclosing scopes, the program fails with an error message.

5 Towards a Dee Style

Any programming language encourages some styles of programming and discourages others. Dee is no exception. As an object oriented language, Dee supports an object oriented style of programming which has been widely discussed [33, 34]. In this section, we discuss some aspects of object oriented programming which have received little attention.

The traditional approach to software development recommends interleaved refinement of control and data structures. Dee, perhaps to an even greater extent than other object oriented languages, encourages a programming style that is independent of data representation. (To avoid the cumbersome phrase “representation independent programming,” we refer to this style of programming as “abstract programming.”) Most of the classes of an application program are clients of abstract classes such as Ring and Collection. The root class, which typically creates the major objects needed by the application, provides instances of concrete classes such as Complex, Matrix, BinaryTree. This style raises two questions which deserve some discussion.

First, we must ask if programs are acceptably efficient. There is no doubt that delayed binding incurs a cost in performance. The cost can be reduced to an acceptable level, however, by global optimization during linking. In any case, it should be clear from previous sections of this paper that efficiency is not the primary goal of our design.

The second and more interesting issue raised by abstract programming is the design of appropriate classes. Class libraries typically provide neatly packaged implementations of conventional data structures such as arrays, lists, stacks, queues, trees, and hash tables. Such libraries support what we have called the “traditional” style of programming, in which representations are chosen relatively early in program development.

Abstract programming requires a hierarchy of abstract classes over the data structure classes. The hierarchy should respect two design criteria. First, an abstract class should not reveal peculiarities of the concrete classes that implement it. Second, a concrete class should not be required to provide unnatural methods merely so that it can inherit from a particular abstract class. The first criterion is conventional but the second requires some elaboration.

A singly-linked list is a useful data structure for many purposes but there is no simple and efficient way to delete an arbitrary component from such a list. Thus, if the class Collection

provides a method `delete`, then the class `SinglyLinkedList` should not be a descendant of `Collection`. There are other aggregating structures for which `delete` is awkward. This suggests that there should be two abstractions for collections: a parent class which does not provide deletion and a child class which does.

A similar situation arises with arrays. Abstractly, an array is a collection indexed by a subrange of the natural numbers or by a set in one-to-one correspondence with such a subrange. The time required to increase the number of components of an array depends on its implementation. Thus the most general array class should not allow the number of components to be increased, but some of its descendants may allow such an increase.

To summarize, an object oriented programming environment must provide a hierarchy of abstract classes supported by another hierarchy of concrete classes. The novelty here is that the structure of these hierarchies is more complex than has previously been suggested.

6 Conclusion

Traditionally, programmers had a choice of two styles of programming. On the one hand, they could have compiling, static analysis, little or no interaction with running programs, and efficiency. On the other hand, they could have interpreting, dynamic analysis, interaction with running programs, and some degree of inefficiency. Languages which supported the first style of programming, such as C and Pascal, offered fast compilers. Languages which supported the second style, such as LISP and Smalltalk, offered rich development environments. In Dee, we have attempted to provide the best of both worlds.

Programmers need rapid access to accurate and up-to-date information about the programs that they are developing. If the languages that they use provide classes and inheritance, they need to know what classes can do and how they are related to one another. The class hierarchy browser of Smalltalk provides some of this information; becoming familiar with it is an essential part of learning Smalltalk. The **short** and **flatten** utilities of Eiffel extract information from source code.

In Dee, we have built on both of these ideas by providing *views*, as explained in Section 3. Because the views are constructed by the compiler, they can contain information derived by the compiler in addition to information provided to the compiler. For example, the client view of a class contains all of the ancestors and all of the inherited attributes of the class. It excludes information that the programmer does not need, such as private attributes. There are advantages in using the compiler to generate information that will be used by the environment. Since the compiler carries out a detailed analysis of the source code anyway, it is natural and efficient to retain the results of the analysis rather than recreate them with other tools.

Object oriented programs typically use many classes, only a few of which are written for a specific application. The designers of Smalltalk realized that programmers would need help to find their way around the system. In designing and implementing Dee, we have transformed the passive concept of “browsing” into the active concept of generating the information that the programmer needs and providing easy access to it.

Acknowledgements. This paper owes much to stimulating discussions with Anne Bennett, Benjamin Cheung, Chris Coyle, Sharon H. Nelson, Markku Sakkinen, Brian Shearing, and Paul

Voda. The Dee project is supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

- [1] 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.
- [2] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
- [3] D. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, December 1972.
- [4] B. Liskov and S. Zilles. Programming with abstract data types. *ACM SIGPLAN Notices*, 9(4):50–9, April 1974.
- [5] J. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.
- [6] P. Grogono. The book of Dee. Technical Report OOP-90-3, Department of Computer Science, Concordia University, February 1990.
- [7] P. Grogono and B. Cheung. Database support for browsing. Technical Report OOP-91-1, Department of Computer Science, Concordia University, January 1991.
- [8] I. Sommerville. *Software Engineering*. Addison-Wesley, third edition, 1989.
- [9] R. Milner. A theory of type polymorphism in programming. *J. Comp. and Sys. Science*, 17:348–375, 1978.
- [10] A. Ohori and P. Buneman. Static type inference for parametric classes. In N. Meyrowitz, editor, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 445–456, 1989.
- [11] 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.
- [12] N. Wirth. *Programming in Modula-2*. Springer, 1982.
- [13] J. Bentley. *Writing Efficient Programs*. Prentice-Hall, 1982.
- [14] B. Meyer. *Object-oriented Software Construction*. Prentice Hall International, 1988.
- [15] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
- [16] P. Grogono and A. Bennett. Polymorphism and type checking in object-oriented languages. *ACM SIGPLAN Notices*, 24(11):109–115, November 1989.

- [17] J. Shilling and P. Sweeney. Three steps to views: Extending the object-oriented paradigm. In N. Meyrowitz, editor, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 353–361, 1989.
- [18] W. Cook. A proposal for making Eiffel type-safe. *Computer Journal*, 32(4):305–311, August 1989.
- [19] B. Meyer. Genericity versus inheritance. In N. Meyrowitz, editor, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 391–405, 1986.
- [20] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
- [21] P. Grogono. Design criteria for a simple object-oriented language. Technical Report OOP-89-5, Department of Computer Science, Concordia University, June 1989.
- [22] D. Moon. The COMMON LISP object-oriented programming language standard. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 4, pages 49–78. ACM Press, 1989.
- [23] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. McQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 51–68. Springer-Verlag, Lecture Notes in Computer Science 173, 1984.
- [24] D. Ingalls. A simple technique for handling multiple polymorphism. In N. Meyrowitz, editor, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 347–349, 1986.
- [25] K. Hebel and R. Johnson. Arithmetic and double dispatching in Smalltalk. *J. Object-Oriented Programming*, 2(6):40–44, March/April 1990.
- [26] D. Ingalls. The Smalltalk-76 programming system: Design and implementation. In *Proc. Fifth ACM Symp. on Principles of Programming Languages*, pages 9–15. ACM, 1978.
- [27] B. Stroustrup. What is “object-oriented programming”? In *European Conf. on Object Oriented Programming*, pages 53–70. Springer LNCS 276, 1987.
- [28] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, 1976.
- [29] B. Liskov, E. Moss, C. Schaffert, and A. Snyder. Abstraction mechanisms in CLU. *Comm. ACM*, 20(8):564–576, August 1977.
- [30] B. Liskov and A. Snyder. Exception handling in CLU. *IEEE Trans. Soft. Engrg.*, SE-5(6):547–558, November 1979.
- [31] B. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [32] A. Koenig and B. Stroustrup. Exception handling for C++. *J. Object-Oriented Programming*, 3(2):16–33, July/August 1990.

- [33] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: an objective sense of style. In N. Meyrowitz, editor, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 323–334. ACM, 1988.
- [34] B. Meyer. From structured programming to object-oriented design: the road to Eiffel. *Structured Programming*, 10(1):19–39, 1989.