**Data & Analysis Center for Software**
**ITT Systems & Sciences Corporation**
Griffiss Business & Technology Park
775 Daedalian Drive
Rome, NY 13441-4909

# An Overview of Object Oriented Design

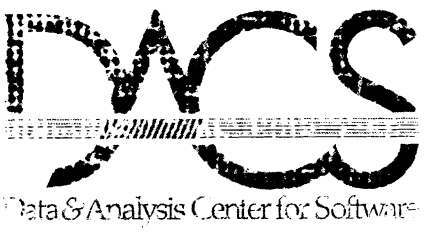
**30 April 1991**

**Prepared for:**

**Rome Laboratory**
**(RL/C3CB)**
**525 Brooks Road**
**Griffiss AFB, NY 13441-4514**

**Prepared by:**

**ITT Systems**
**(formerly Kaman Sciences Corporation)**
**P.O. Box 1400**
**Rome, NY 13442-4905**


The Data & Analysis Center for Software (DACS) is a Department of Defense (DoD) Information Analysis Center (IAC), administered by the Defense Technical Information Center (DTIC), Fort Belvoir, Virginia), technically managed by Air Force Research Laboratory (AFRL), Rome, New York. ITT Systems & Sciences Corporation manages and operates the DACS, serving as a source for currently readily available data and information concerning software engineering and software technology.

# REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188



| 1. AGENCY USE ONLY (Leave blank) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | April 30, 1991 | N/A |

**4. TITLE AND SUBTITLE**

An Overview of Object Oriented Design

**5. FUNDING NUMBERS**

F30602-89-C-0082

**6. AUTHOR(S)**

Robert Vienneau

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

Kaman Sciences Corporation
258 Genesee Street
Utica, NY 13502

**8. PERFORMING ORGANIZATION REPORT NUMBER**

N/A

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

Defense Technical Info Ctr
DTIC/DF
Cameron Station
Alexandria, VA 22314

Rome Laboratory
RL/COEE
Griffiss AFB, NY 13441

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

N/A

**11. SUPPLEMENTARY NOTES**

Available from: Data & Analysis Center for Software (DACS)
258 Genesee Street
Utica, NY 13502 Phone (315) 734-3696

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**

Approved for public release, distribution unlimited

**12b. DISTRIBUTION CODE**

**13. ABSTRACT (Maximum 200 words)**

This report, An Overview of Object Oriented Design, provides a basic understanding of Object Oriented Design (OOD) and some of its features. The report briefly summarizes the history of OOD, includes a description of an OOD methodology, and defines and discusses various concepts and terminology used in OOD. The level of support that various programming languages provide for OOD is discussed in some detail. Languages covered include Modula-2, Ada, C++, Object C, LISP, Smalltalk, and Eiffel. Section 4 discusses how OOD interacts with areas of current software engineering research, especially software reuse and alternative life cycle models. The report also includes a glossary of OOD terms and an annotated bibliography of related papers and reports.

**14. SUBJECT TERMS**

Object Oriented Design, Computer Software, Computer Programming, Software Technology, Software Engineering.

**15. NUMBER OF PAGES**

83

**16. PRICE CODE**

N/A

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | UL |

# TABLE OF CONTENTS

# FOREWORD

This report, *An Overview of Object Oriented Design*, provides a basic understanding of Object Oriented Design (OOD) and some of its features. The report briefly summarizes the history of OOD, includes a description of an OOD methodology, and defines and discusses various concepts and terminology used in OOD. The level of support that various programming languages provide for OOD is discussed in some detail. Languages covered include Modula-2, Ada, C++, Objective C, LISP, Smalltalk, and Eiffel. Section 4 discusses how OOD interacts with other areas of current software engineering research, especially software reuse and alternative life cycle models. The report also includes a glossary of OOD terms and an annotated bibliography of related papers and reports.

# 1. INTRODUCTION TO OBJECT ORIENTED DESIGN

Object Oriented Design (OOD) grew out of work in both design methodology and language design. OOD was developed to provide a natural method of structuring system architectures. The resulting architecture can be easily changed. Module interfaces (should) prevent changes from rippling through the system. Language concepts originally developed to support simulation were seen to provide features that could easily be used to implement such an architecture. The integration of these insights lays the foundation for OOD.

The 1980s have seen an explosion of interest in Object Oriented Design among professional programmers. This is evident in the history of the Object Oriented Programming Systems, Languages, and Applications (OOPSLA) conferences. The first such conference was organized by the Association of Computing Machinery (ACM) in 1986. By 1988, only two years later, OOPSLA was the third largest of ACM's many technical conferences.

Within the defense community, an interest in Object Oriented Design (OOD) is almost mandated by the use of Ada. Ada is more than a programming language; proper use of Ada requires the adoption of various software engineering techniques, including OOD. On the other hand, some OOD purists argue that Ada does not truly support OOD. This issue, among others, will be explored in this report.

Among the wider community of programmers and computer enthusiasts, an interest in OOD is demonstrated by a plethora of products. For example, Borland International and Microsoft Corporation, the major distributors of compilers for personal computers, both market Object Oriented versions of several languages, such as Pascal. Graphical user interfaces, such as the Macintosh interface, have been described as Object Oriented, although purists argue otherwise.

Certainly, OOD is beginning to influence many areas of computer science. OOD is currently important in research in reusability, user interfaces, Ada, requirements and design methodologies, programming environments, and Data Base Management Systems (DBMS). OOD proponents claim OOD will dramatically increase the productivity and quality of software. For example, Brad Cox compares OOD to the introduction of interchangeable parts into production [Cox 1986]. Just as that innovation made possible mass production and "revolutionized manufacturing forever," so OOD promises to transform the process of programming from the churning out of thousands of lines of code to the assembling of systems from libraries of predefined modules.

## 1.1 Origins of OOD

The groundwork of Object Oriented Design was laid concurrently with the introduction of structured programming in the 1960s. Certain languages developed at that time provided features that would become basic elements of OOD.

In particular, Simula[1], a block structured language with a main program and nested entities such as subroutines, introduced classes, now thought to be key for OOD. A class, in Simula, is similar to a data type. A class may contain routines, attributes, and instructions that are automatically executed when an instance of that class is created. Variables of user-defined types reference instances of a class. Instances of a class later came to be known as objects, the objects that OOD gets its name from.

The concepts introduced by Simula did not gain wide currency for decades, even among researchers. This lack of circulation may be largely because Simula was a solution in search of a problem. Those features of Simula that support OOD were not clearly seen as supporting a general purpose system design methodology. Rather, they were seen as supporting simulation, their original justification for being introduced.

The structured programming revolution was barely begun in 1972 when David Parnas began criticizing its inadequate support for modularizing systems. In his "On the Criteria to be Used in Decomposing Systems into Modules," he presents an example Keyword In Context (KWIC) problem and contrasts two methods of modularizing the solution [Parnas 72].

The first solution, which might result naturally from top down design, contains input, circular shift, alphabetizing, and output modules. This modularization results from considering the steps needed to solve the problem. The second solution results from considering the data structures needed to solve the problem. Each module provides access to a different data structure[2]. Unlike modifications to the first solution, changed functionality in the second solution will typically result in modifications to only a couple of modules. The interfaces prevent changes from rippling through the system. Furthermore, the sequence of instructions necessary to call a given routine and the routine itself are part of the same module. Parnas' ideas inspired research embodied in CLU, Modula-2, and Ada, programming languages introduced in the late 1970s and early 1980s.

---

[1]Simula, sometimes known as Simula-67 after the year in which it was introduced, was designed by Ole-Johan Dahl and Krysten Nygaard at the University of Oslo and the Norwegian Computing Center. Simula is an extension of Algol 60 intended to provide strong support for computer simulation.

[2] In short, Parnas introduced the important concept of "information hiding" in which frequently changed components of a system are hidden in single modules, not scattered through many modules nor part of the module interfaces. Parnas argues that modularization on the basis of data structures will allow programs to be easily changed, modules to be developed independently, and the system to be more easily understood.

Parallel to these developments came Smalltalk[3], a language that built on Simula's innovations and further refined OOD. Smalltalk borrowed heavily from Simula, but introduces some new notions such as an advanced implementation of inheritance, polymorphism, and dynamic binding. Variables can take on many types; ambiguities are resolved at run time. In many ways, Smalltalk was the purest Object Oriented language available in the 1980s.

More recently, the focus of OOD is on the integration of these separate strands of design methodology, modularization, and programming languages. Russell Abbott and Grady Booch have developed a OOD methodology for Ada, based in part on insights derived from Smalltalk ([Abbott 83], [Abbott 86], [Booch 86], and [Booch 87b]). Others have attempted to integrate Object Oriented concepts with current practice not by developing a methodology for widely used languages, but by extending current languages with Object Oriented constructs. Finally, Bertrand Meyer has introduced Eiffel, a pure Object Oriented language intended to be less research oriented than Smalltalk.

Efforts to extend OOD continue. Researchers have begun to explore the role of "persistent objects," objects that remain after a program's execution has ceased. These ideas have inspired Object Oriented Databases, created as an alternative to the traditional hierarchical, relational, or entity-relationship models. Much current research on user interfaces has an Object Oriented flavor. Researchers are also studying how to smoothly integrate concurrency into Object Oriented programming languages.

[3] Smalltalk, was developed by Alan Kay, Adele Goldberg, and Daniel H. H. Ingalls at the Xerox Palo Alto Research Center (PARC).

# 2. METHODOLOGY

The architecture of systems constructed with OOD is based on objects, not functions. Other design methods, most notably top down design, focus on the function of the software.[4] Although Object Oriented systems are structured around data, an object is something more than a data structure. Grady Booch defines an object [Booch 86] as an entity that:

- has a state
- is characterized by the operations that can be performed on it and that it requires to be able to perform on other objects
- is denoted by a name
- is an instance of a class
- can be viewed by its specification as well as by its implementation

In a sense, the state of an object is analogous to the value of a variable. However, important differences exist between these concepts. Two objects can be represented by data with different values and yet have the same state. For example, if a stack is implemented by an array and an index to the top of the stack, two stacks might differ in the values of those array elements which denote items that are currently not members of the stack. Though represented by arrays with different values, these stacks still have the same state, as shown in Figure 2.1.

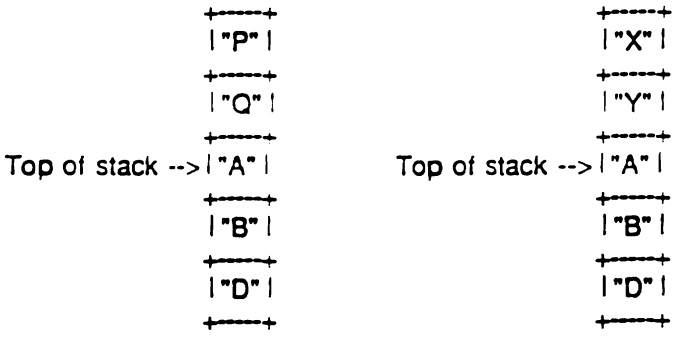



FIGURE 2.1: TWO STACKS WITH THE SAME STATE

[4] Bertrand Meyer, the designer of the Object Oriented programming language Eiffel, argues [Meyer 88] that large systems typically cannot usefully be characterized as performing a single function. Rather, they perform an interrelated group of functions. Operating systems and Management Information Systems are good illustrations of this thesis originally propounded by David Parnas [Parnas 72].

Pointers provide another familiar example of variables with different values, but the same state. In certain applications, two pointers have the same state if they access memory locations where equal values are stored, even if the pointers differ in the addresses to which they point.

OOD supports the concept of information hiding by distinguishing between data values and the state of an object. Only aspects relevant to the state of an object should be made available in its specification.

Objects can access other objects by the operations made available in the specification of the accessed objects. Booch 87a, Meyer 88, and Stroustrup 88 classify these operations as either constructors, destructors, selectors, or iterators.

Constructors and destructors modify objects: constructors either create an object or modify its state while destructors destroy objects. The environment provided with some Object Oriented languages performs automatic garbage collection. Hence, destructors are not needed in these languages, a great convenience to programmers. The automatic invocation of certain programmer-defined constructors in some languages when an object is declared is another convenience. This feature allows certain invariants to be automatically established.

Selectors and iterators are used to obtain information about the state of an object. For example, an operation that returns the top of a stack is a selector, while operations that allow one to obtain the value of every node in a tree are iterators. Iterators permit all parts of an object to be visited.

OOD has not yet developed a uniform terminology. Hence, not all writers and language designers use the terms constructor, destructor, selector, and iterator.

The distinction between classes and objects also varies among Object Oriented systems. In all cases, an object is an instance of a class. An object might be a specific stack of characters; the corresponding class is the class of all such character stacks. Object oriented systems allow the programmer to provide names to refer to the objects. A further distinction arises between objects and classes. In Eiffel classes are defined in the program text; they do not exist at run-time [Meyer 88]. Objects, however, only exist at run-time.[5]

The manner in which objects perform operations on each other suggests one type of relation between classes. Classes can form client-supplier pairs. A supplier provides operations which other classes can use. The classes that use these operations are known as clients of the supplier. Classes can also relate to one another by forming ancestor-descendant pairs. A descendant class is an extension or specialization of its ancestor. The concept of descendant derives from the concept of inheritance, which some argue is crucial for OOD.

[5] Smalltalk does not make this distinction; Smalltalk classes can exist at run-time.

## 2.1 Abstract Data Types (ADTs)

Classes can be regarded as implementations of Abstract Data Types (ADTs) ([Meyer 88] and [Booch 87b]). An ADT defines the formal properties of a data type without defining implementation features. Hence, ADTs are one mechanism of formalizing the concept of information hiding. ADTs are specified in formal languages, like those studies in mathematics, especially the predicate calculus, rather than the traditional imperative programming languages.[6]

For example, an ADT specification for a queue can be represented as shown in Figure 2.2.

The first part of the specification defines the data types for the user. This is a parameterized type; X is the type of the elements comprising the queue. Note that the specification is totally implementation independent. Whether a queue is a linked list, an array, or not implemented at all is not shown here.

The second section in the specification lists the functions or operations available. The notation is borrowed from advanced mathematics. Only the syntax of the operations is given, not their meanings. Each function maps a domain to a range with these sets on either side of the arrow. The function "new" has an empty domain as is shown by a null left side. The function "enqueue" takes two arguments as shown by the cartesian product on the left side.

The preconditions and the axioms give the semantics of a queue. The preconditions show the subset of the domain of all queues to which the functions can be applied. The axioms characterize the behavior of queues. The first element of an empty queue is not defined. Neither is the result of dequeuing an empty queue. Consequently, the functions "first" and "dequeue" are only partial functions. The first axiom shows that a new queue is empty, while the second axiom shows that enqueuing an element results in a non-empty queue. The final three axioms characterize how items are enqueued and dequeued. The notation $f^n(x)$ means apply the function n times in succession to the variable x. Consequently, the third axiom shows the first n elements of an n element queue are unchanged by enqueuing an element. The fourth axiom shows that enqueuing an item adds it to the end of the queue. Finally, the fifth axiom shows that enqueuing an item increase the number of items in a queue by one.

Once this formalism is used to specify an ADT, it can be used to reason about programs. For example, consider the following program fragment:

[6] ADTs predate OOD; they were an intermediate phase between Parnas' modularization criteria and OOD.

TYPES

Queue[X]

FUNCTIONS

is_empty: Queue[X] --> Boolean

new: --> Queue[X]

enqueue: X x Queue[X] --> Queue[X]

dequeue: Queue[X] --> Queue[X]

first: Queue[X] --> X

PRECONDITIONS

pre dequeue(q: Queue[X]) = not is_empty(q)

pre first(q: Queue[X]) = not is_empty(q)

AXIOMS

for all x: X, q: Queue[X]

1. is_empty(new())
2. not is_empty(enqueue(x, q))
3. if is_empty(dequeue$^n$(q)) then, for all k < n,

   first(dequeue$^k$(q)) = first(dequeue$^k$(enqueue(x, q)))
4. if is_empty(dequeue$^n$(q)) then

   first(dequeue$^n$(enqueue(x, q))) = x
5. if is_empty(dequeue$^n$(q)) then

   is_empty(dequeue$^{n+1}$(enqueue(x, q)))

FIGURE 2.2: AN ADT FOR A QUEUE

```
q := new();
enqueue(a, q);
dequeue(q);
```

After enqueuing and dequeuing a single item on a new queue, the resulting queue should be empty.
The first axiom states

```
is_empty(new())
```

Consider the fifth axiom with n set equal to 0. By convention, applying a function zero times is equivalent to applying the identity function. Hence, the fifth axiom implies

```
if is_empty(new()) then is_empty(dequeue(enqueue(a, new())))
```

By inference, one can conclude

```
is_empty(dequeue(enqueue(a, new())))
```

This proves that the code fragment results in an empty queue.

This exposition demonstrates that object-oriented design is the construction of software systems as structured collections of Abstract Data Type implementations [Meyer 88]. ADTs are implemented as classes. An Object Oriented system is a collection of classes. The collection is structured by the supplier-client relationship and by inheritance.

## 2.2 Designing Object Oriented Systems

According to Grady Booch, OOD is performed in five steps: [7]

1. Identify the objects
2. Identify the operations
3. Establish visibility
4. Establish interfaces
5. Implement each object.

[7] The broad outline of the methodology presented here is based on that developed by Grady Booch in various books and papers ([Booch 86], [Booch 87a], [Booch 87b]). Discussion of the details of various steps draws on work by Russell Abbott ([Abbott 83], [Abbott 86]) and Bertrand Meyer ([Meyer 88], especially chapter 14).

### 2.2.1 Identify the Objects

Identifying the objects is actually performed by identifying the classes of which objects are members. The physical system that the software is modeling should provide some obvious indications of candidate objects: a physical object might provide a software object. For example, if a window-based user interface is being designed, potential objects are mice, keyboards, screens, menus, and windows.

Libraries of classes may exist which can provide the designer with candidate objects. Aside from queues, stacks, linked lists, hash tables, trees, and other data structures in computer science applications, these libraries might include application-oriented objects. For example, the Common Ada Missile Packages (CAMP) includes Ada packages for Kalman filters, radars, and altimeters [CAMP 87].

Inheritance makes possible the existence of objects that may be very hard to identify. If several classes share similar attributes or operations, a good design might abstract these general properties into a new class. The original classes will inherit these properties from this new class. In fact, the new class might be inapplicable in its own right; its only purpose for existence could be to provide properties that more usable classes could inherit.

As a design progresses, one may identify additional objects. The decentralized nature of OOD should allow such objects to be easily added. Hence, OOD attempts to promote the learning process that naturally occurs when designing any software system.

### 2.2.2 Identify the Operations

Objects often can be modeled as ADTs which should suggest operations. For example, a queue might have operations for checking whether it is empty, creating an empty queue, enqueuing an element, dequeuing an element, and returning the first element. Objects can also be modeled by means of Abstract State Machines, their use suggesting operations. Such operations will include returning information on the current state and switching from state to state.

Finally, one may think of the operations available on an object as a shopping list. It is allowable for operations to be made available that no client buys (uses). Too many operations for a given class, however, may suggest that the class be broken up into several classes.

### 2.2.3 Establish Visibility

Once the classes, objects, and operations are determined, relationships must be established among them. Objects can relate in terms of either client-supplier or ancestor-descendant pairs. The decision whether a class should be an heir or a client of another class requires judgment.

More research needs to be done on how to structure relationships between classes. Some Object Oriented systems have no structure level at a higher level than classes. Yet a large system may have on the order of hundreds of classes at the same level, a questionable design.

Various ideas on the architecture of Object Oriented systems have been proposed. Some have suggested that Object Oriented systems be developed as layers of virtual machines (eg. Smalltalk includes a metaclass as a higher level structuring mechanism [Cointe 87]). Some have objected to inheritance, proposing various relationships between objects to supersede or control it ([Minsky 87], [Stein 87], and [Lieberherr 89]).

### 2.2.4 Establish Interfaces

The step of establishing interfaces amounts to formally describing the public view of an object. Ada package specifications, which serve this purpose, can be programmed and compiled[8] before the corresponding package bodies are coded. Modula-2 provides a similar facility. Both the Ada package specification and the Modula-2 definition module establish what is available to users of an object. Package specifications are also useful as a project management tool; they allow many programmers to work in parallel on the package bodies with minimal communication.

### 2.2.5 Implement Each Object

The final step in the OOD methodology is to actually implement each object. Principles of structured coding should be used for the detailed design of each operation. Only a limited number of constructs (sequence, iteration, selection) should be used, and in a manner that permits formal verification.

This step can actually be argued to belong to the coding phase. In some sense, OOD decreases the distinction between design and coding. The goal is to provide a natural method of representing characteristics of the problem domain in source code. Inasmuch as OOD achieves this goal, the transition from requirements to source code should not result in an abrupt change in technology at the end of the design phase and the beginning of the coding phase.

[8] Some Object Oriented languages do not provide for separately compilable interfaces. Eiffel provides for deferred classes, which serve the same purpose.

# 3. LANGUAGE SUPPORT FOR OOD

Object Oriented Design is a methodology that was developed in the context of various programming languages. The interactions of objects can make demands of the language, as well as the development environment itself. This section defines a scale for measuring the degree to which a language supports OOD and actually compares various languages on the basis of the scale.

## 3.1 A Scale For Measuring Object Oriented Languages

Bertrand Meyer has proposed seven levels[9] that measure how well a programming language supports OOD.

1. Modularization on the basis of data structures
2. Ability to describe objects as implementations of Abstract Data Types (ADTs)
3. Automatic garbage collection
4. Equivalence of modules and (non simple) types
5. Inheritance
6. Polymorphism and dynamic binding
7. Multiple and repeated inheritance

(1) **Modularization on the basis of data structures** means the language allows the type declarations, constants, and procedures pertaining to a particular data structure to be grouped together. The collection of such declarations should be capable of being set off from the rest of the program in a separate file or module.

(2) **Data abstraction** is possible if the language allows the specification of a data structure or object to be separated from its implementation. This separation should allow the specification to be regarded as an implementation of an Abstract Data Type. The language should automatically prevent unauthorized access to implementation details that the programmer wants hidden.

(3) **Automatic garbage collection** is an implementation feature convenient for programmers. Garbage collection occurs when the underlying language system automatically reclaims space occupied by inaccessible data items.

[9] Brad Cox, the inventor of Objective C, compares OOD languages by means of eight characteristics, four in Meyer's scale, the other four being commercial availability, the availability of libraries, and two technical properties related to polymorphism [Cox 86].

(4) **Equivalence of modules and types** exists in languages that do not distinguish between the declaration of types and modules. Meyer defines a language construct combining module and type aspects as a class. Variables of such types are known as objects.

(5) **Inheritance** allows a class to be defined as an extension or specialization of another class. Inheritance resembles subtyping in more traditional languages. As an example of inheritance, consider a system for a windowing user interface. For example, in a system for a windowing user interface, a class of geometric shapes might be defined. Typically, operations would return the center and color of a shape, expose it, hide it, save it, move it, and so forth. Classes for special shapes such as triangles, squares, and circles would include specialized operations for areas and perimeters, as well as all of the operations of a general geometric shape. Inheritance provides for implementing these classes without duplicating the code for operations that apply to general shapes. The special shapes then are declared as descendants of the class of general geometric shapes having all the properties and operations of a shape, as well as their own special operations.

(6) **Polymorphism and dynamic binding** identifies the ability of an object to be a member of different classes at different points of time during the execution of a system. In more traditional terminology, polymorphism is the ability of a variable to take on more than one type during execution. Dynamic or late binding allows the version of an operation which will be applied to an object to be chosen as late as run time. Ordinarily, early binding results from a decision on the version of an operation to be applied at compile time; polymorphism prevents this. Polymorphism and dynamic binding support software development by allowing source code to properly localize certain design decisions, thus leading to more understandable and easily modified code.

(7) **Multiple inheritance** is achieved when a class can inherit properties from more than one class with the possibility of inheriting properties from another class in more than one way. As an analogy, consider the case where a father and mother are distant cousins. Any children of such an union can refer to some of their ancestors under more than one description. Designers of Object Oriented languages that support multiple and repeated inheritance must resolve any ambiguities on the interpretation of an operation that may arise from repeated inheritance. Consider, for example, two classes: one for integers and another for arrays. By defining a new class that inherits from both these classes, one can create a class for arrays of integers [Wiener 89]. This example demonstrates that multiple inheritance can be at least as powerful as generics, although maybe not always as convenient.

Current languages tend to cluster in three locations on this scale (Table 3.1). First, classical languages, such as Assembly, Basic, C, Fortran, Jovial, and Pascal, do not necessarily allow programmers to modularize on the basis of data structures. Modern data abstraction languages such as Ada and Modula-2 provide powerful data abstraction and information hiding facilities that allow one to

| LEVEL: | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Fortran | | | | | | | |
| Pascal | | | | | | | |
| C | | | | | | | |
| Modula-2 | A | A | | | | | |
| Ada | A | A | I | D | D | | |
| C++ | A | A | D | A | A | A | A |
| Objective C | A | A | A | A | A | A | A |
| Lisp extensions | A | D | A | A | A | A | |
| Smalltalk | A | A | A | A | A | A | A |
| Eiffel | A | A | A | A | A | A | A |

**Key:**

**Level:**
1 = Modularization on the basis of data structures
2 = Data abstraction
3 = Automatic garbage collection
4 = Module-type equivalence
5 = Inheritance
6 = Polymorphism and dynamic binding
7 = Repeated inheritance

A = Level achieved
I = Achieved in some implementations
D = Level achieved, but differences noted

TABLE 3.1: PROGRAMMING LANGUAGE SUPPORT OF OOD, SCALE SUMMARY

implement ADTs. Yet they do not fully support OOD. Neither do they support polymorphism, dynamic binding, and inheritance, nor do they allow a type to be identified as a module. Finally, full Object Oriented languages that lie on the upper end of the scale may be partitioned into two groups: extensions of classical languages (C++, Objective C, Flavors, and Loops) and pure object oriented languages (Smalltalk and Eiffel).

## 3.2 Classical Languages

In general, Algol style block structured languages fail to even reach the first level on this scale. Examples of such languages widely used include Fortran, Pascal, and C. The primary structuring mechanism in such languages is the procedure or subroutine. Some awkward implementation techniques have been discovered for simulating certain OOD concepts such as classes and inheritance in these languages [Meyer 88]. These techniques, however, violate the spirit of the language and result in code that most conventional programmers would find fairly strange. In general, although one can certainly implement an Object Oriented Design in these classical languages, the resulting code fails to reflect the structure of the design.

## 3.3 Data Abstraction Languages

Two influential languages, Ada and Modula-2, introduced during the early eighties, reflected much research in language design. Although OOD was already used in certain small research communities by the time these languages were designed, it had yet to gain the popularity it currently enjoys. Consequently, these languages do not fully implement some Object Oriented concepts.

### 3.3.1 Modula-2

Modula-2 reaches the second level on the scale. It allows objects to be described as implementations of ADTs, but it does not provide for automatic garbage collection, nor for the identification of types and modules.

Modula-2 provides a method of modularization above the procedure level and more rigorous than that provided by an operating system's file structure. This method, as provided by Modula-2 modules, can be used to modularize on the basis of data structures. Modula-2 also allows the definition of a module to be separated from its implementation. The resulting modules can be viewed as implementations of Abstract Data Types. For example, the definition module of a queue might look like the following:

```
DEFINITION MODULE Queue;
EXPORT QUALIFIED Queue, New, IsEmpty, Enqueue, Dequeue, First;
  (* Comments can provide axioms for these procedures. *)
```

```
TYPE Queue;          (* Queue of CARDINALs. *)
PROCEDURE New(VAR q: Queue);
PROCEDURE IsEmpty(q: Queue): BOOLEAN;
PROCEDURE Enqueue(x: CARDINAL; VAR q: Queue);
PROCEDURE Dequeue(VAR q: Queue);
PROCEDURE First(q: Queue): CARDINAL;
END Queue.
```

Unlike most languages, Modula-2 is case sensitive; all keywords must be uppercase. This restriction almost mandates a certain style for user defined entities. Note the declaration of Queue in the the above definition module for a queue. Modules and types are distinct in Modula-2, but this example demonstrates the utility of a mechanism for combining these two concepts. Both the module and the type share the same name. Queue is declared as an "opaque type," Modula-2 jargon for types made available in a definition module, but implemented in an implementation module. Modules that import Queue may only reference it by means of the operations exported from the queue module. Hence, Modula-2 can be used to enforce information hiding and data abstraction techniques. In many implementations, opaque types suffer from a limitation; they can only be one word long. This restriction is overcome for objects, such as queues, that require many words of storage by implementing such objects as pointers. The pointer itself will be a word of storage, but the entity that it points to can be any length. Thus, to use Modula-2 to implement data abstraction techniques requires considerable use of dynamically allocated memory. Since Modula-2 does not provide for automatic garbage collection, the programmer must manage the heap himself. Care is required in preventing aliasing and in allocating and deallocating memory.

### 3.3.2 Ada

Ada does not fully nor completely support OOD. Ada definitely allows objects to be described as ADTs. Implementations are permitted to provide automatic garbage collection. Multitasking applications can regard Ada tasks as both modules and types. Ada definitely does not provide polymorphism and dynamic binding. Nevertheless, Ada introduces many powerful new features (whose capabilities are still a matter of debate). OOD methodologies are widely known in the Ada community, are often taught with introductory Ada courses, and are used by the NASA Goddard Space Flight Center and other designers of large systems.

Ada reaches, at least, the second level on our scale. Ada package specifications, like Modula-2 module definitions, allow objects to be described as implementations of ADTs. For example, the Ada package specification for a queue might look like the following:

```
generic
   type XType is private;
package Queue is

   type Queue;

   procedure NEW(Q: in out Queue);

   function IS_EMPTY(Q: in Queue) return Boolean;

   procedure ENQUEUE(X: in     XType; Q: in out Queue);
   -- Raises OVERFLOW if no more space can be allocated for Q.

   procedure DEQUEUE(Q: in out Queue);
   -- Raises UNDERFLOW if IS_EMPTY(Q).

   function FIRST(Q: in Queue) return XType;
   -- Raises UNDERFLOW if IS_EMPTY(Q).

   OVERFLOW : exception;
   UNDERFLOW: exception;

private

   type Queue_Node;
   type Queue_Pointer is access Queue_Node;
   type Queue_Node is record
      X    : XType;
      NEXT: Queue_Pointer;
   end record;

   type Queue is record
      NUMBER_IN_QUEUE: Natural;
      THE_QUEUE       : Queue_Pointer;
   end record;

end Queue;
```

This example illustrates several differences between Ada packages and Modula-2 modules. First, the queue defined here is generic, with the type of items stored in the queue possibly differing among queues. No comparable feature is provided by Modula-2; the Modula-2 example queue could only contain cardinal numbers. Second, Ada provides user-defined "exceptions," raised when anomalous conditions arise. Control is returned to the calling procedure, but not necessarily at the point of invocation. In effect, exceptions allow multi-exit subroutines. Finally, packages that make use of queues cannot access any entities declared in a private part (see example), but the compiler knows how much space to allocate for queues because of the information in the private part. In effect, information in the private part is invisible to users of queues.

Garbage collection is an implementation decision for Ada compilers[10]. Since implementations can refuse to provide automatic garbage collection, programmers need to be able to explicitly control the allocation of memory. The pragma CONTROLLED and the standard procedure UNCHECKED_DEALLOCATION are used for this purpose.

[10] The Ada Language Reference Manual [ANSI 83] states "An implementation may (but need not) reclaim the storage occupied by an object created by an allocator (pointer), once this object has become inaccessible (Section 4.8)."

The next level on the scale, the equivalence of modules and types, is also partially introduced in Ada. Ada packages are certainly not types, as the Queue example illustrates. However, Ada tasks can be thought of as objects with operations implemented as task entries. Tasks are a natural way to modularize multitasking applications. Moreover, tasks can be explicitly declared as task types. A task can have multiple entries, each providing some service. Consequently, Ada tasks provide features of both modules and types. For Object Oriented language designers to adopt the Ada tasking model, Ada tasks must be shown to be capable of simple and efficient implementation.

Inheritance is probably the most controversial requirement in considering Ada's support for OOD. Ada has a complicated type structure with derived types and subtypes that can simulate inheritance [Perez 88]. Ada's ability to define generic packages and procedures, however, is most closely related to inheritance. Both generics and inheritance are mechanisms for making modules more reusable and flexible[11].

Ada definitely does not support polymorphism and dynamic binding, the next level on the scale for measuring language support for OOD. Ada is strongly typed with binding performed at compile time. Ada does allow liberal use of overloading; many procedures can be given the same name. In any given case, Ada compilers decide based on the number and type of arguments what procedure is being referenced.

## 3.4 Object Oriented Extensions of Classical Languages

As OOD has become more popular, various classical languages have been extended to make OOD more accessible to the broad mass of programmers. Traditional programmers should feel comfortable using them to gradually evolve to Object Oriented Design. Portions of a system, for that matter, can comfortably reflect classical designs and yet be fully integrated with Object Oriented portions.

### 3.4.1 C Extensions

Researchers have recently grafted Object Oriented concepts to a C base. The two most well-known languages resulting from these attempts are C++ and Objective C.

[11] In fact, programmers have found generics so desirable that they have attempted to implement generics in Modula-2 ([Reynolds 87] and [Wiener 85]). Bertrand Meyer concluded that the applications supported by generics are only a proper subset of those supported by inheritance. Some applications, however, are most conveniently supported by generics. Inheritance is more powerful, but a well designed language might also support certain aspects of generics [Meyer 86].

#### 3.4.1.1 C++

C++[12] reaches the upper reaches of the scale for measuring language support for OOD. The programmer can introduce classes which combine types and functions. As in Ada, these classes can have both a public and a private part, a specification and an implementation. For example, the specification of a queue class might be given by the following:

```
class QUEUE {
private:
  int thefirst, thelast, impl[MAXSIZE];
public:
  void new();
  BOOLEAN isempty();
  void enqueue(integer);
  void dequeue();
  int first();
};
```

Not shown in this example are friend functions, a new concept introduced in C++. Unlike the member functions defined above, friend functions require an argument denoting the object to which they are applied. Friend functions allow C++ functions to be called from normal C code.

C++ has inheritance, polymorphism, and dynamic binding. Multiple inheritance was introduced in Version 2. C++ does not provide automatic garbage collection, but the ability to automatically invoke destructors removes much of the burden associated with memory deallocation.

#### 3.4.1.2 Objective C

Objective C[13] is a preprocessed extension to C that fully supports OOD. It provides automatic garbage collection, polymorphism, dynamic binding, and inheritance (including multiple inheritance). Objects are supported by means of a new data type, an object identifier. Just as variables can be declared as belonging to a C base type, so variables can be declared as an object identifier in Objective C. A variable of the object identifier can be used to hold an identifier for any type of object.

### 3.4.2 Lisp Extensions: Flavors and Loops

OOD might be regarded as a small twist to traditional procedural oriented programming. Instead of regarding procedures as acting on data, OOD adopts the view that objects invoke methods in response to messages, an important new method of designing system architectures. At the lowest level,

[12] C++ was developed beginning in 1980 by the same group at Bell Laboratories that designed C. Interestingly, Bjarne Stroustrup, the inventor of C++, states he was inspired to develop C++ partly to develop a simulation program, the application area of Simula. Furthermore, Stroustrup's experience with Simula exposed him to the power of Object Oriented languages [Stevens 89].

[13] Objective C was designed by Brad Cox, now with the Stepstone Corporation, drawing on his experience with Smalltalk. Because Dr. Cox is not affiliated with AT&T, he felt himself less free to modify the base language when grafting Object Oriented concepts onto C than was done in C++.

though, traditional structural methods are used, at least in data abstraction languages and Object Oriented extensions to classical languages.

The fact that researchers have added Object Oriented extensions to Lisp, however, suggests that OOD is a dramatic revolution. Lisp[14] has evolved out of a completely different tradition than block structured languages such as Fortran, C, Pascal, and Ada. Lisp is ubiquitous in AI research, but rare elsewhere. It promotes a style known as functional programming [Backus 78]. AI researchers to implement a wide variety of tools and environments in Lisp. Lisp encourages programmers to disregard the distinction between programs and data and write short, highly recursive programs. As part of this style, Lisp programs are weakly typed.

The most popular Lisp extensions supporting OOD are most likely the languages Loops and Flavors[15]. These Lisp extensions give very complete support to OOD, all provide classes with aspects of both types and modules. These classes, known as flavors in Flavors, provide a mechanism to modularize on the basis of data structures. However, consistent with Lisp traditions, the resulting data structures differ from those used in traditional block structured languages. Lisp data structures tend to be recursive lists of lists. Accordingly, natural axioms for describing classes as Abstract Data Types may differ.

Lisp itself relies heavily on automatic garbage collection, with Lisp programmers likely to work in very advanced programming environments. Pure Object Oriented languages are also characterized by advanced environments, but with somewhat different features. Garbage collection is only a small aspect of such environments.

Object Oriented extensions to Lisp support polymorphism, dynamic binding, and inheritance. Dynamic binding is another Object Oriented feature that's fairly trivial to provide to Lisp extensions. Lisp environments are usually interpretive, with many dynamic properties. The weak typing provided by Lisp has important implications for how inheritance is used. The resulting class hierarchies in, say, a Flavors system can be quite different than those supported by a comparably strongly typed language such as C++ [Wolf 89].

Despite the high support for OOD of these Lisp extensions, adopting any one of them should be carefully considered. Adopting a Lisp extensions would involve learning two new paradigms, not just one. The interaction of these paradigms would invariably destroy the possibility of isolating the effects of just one, OOD, on any resulting projects.

---

[14] John McCarthy introduced Lisp in the late 1950s to support research in Artificial Intelligence (AI). Essentially, Lisp is an implementation of Alonzo Church's lambda calculus, introduced to investigate certain theoretical questions in computability and logic. Related work stems from Turing Machines and Godel's incompleteness theorems.

[15] Loops was developed at Xerox, originally as an Interlisp variant. Flavors was developed at the Massachusetts Institute of Technology.

## 3.5 Pure Object Oriented Languages

Pure Object Oriented languages represent the state of the art of OOD. Naturally, these languages also fall in the upper end of the scale.

### 3.5.1 Smalltalk

Smalltalk[16] almost defines the OOD paradigm and lies at the highest end of the scale for measuring Object Oriented languages. Classes and objects provide a natural basis for modularization. Smalltalk provides garbage collection, polymorphism, dynamic binding, and inheritance (including multiple inheritance). Smalltalk provides a good commercially available means of investigating OOD, but it is most widely viewed as a tool useful in producing prototypes, not in developing production system.

### 3.5.2 Eiffel

Eiffel is probably the most well known pure Object Oriented language introduced during the latter half of the 1980s. Since Bertrand Meyer designed Eiffel as well as the scale used to evaluate OOD language support, Eiffel naturally is at the top of the scale.

Eiffel supports all OOD concepts discussed. The basic modularization mechanism, the Eiffel class, combines properties of both modules and types. An Eiffel system is a collection of classes; Control is decentralized. In particular, an Eiffel program contains no main procedure, rather the user declares one class to be the root. This class will create other classes and may pass control to them. Classes can relate to one another through inheritance, including repeated and multiple inheritance. Polymorphism and dynamic binding are provided.

Eiffel borrows certain ideas from Ada including the ability to distinguish between class specifications and implementations, generic classes, and exceptions. Some differences between the use of these features in Eiffel and Ada are illustrated by the example of a queue:

```
class interface QUEUE[XTYPE]
exported features

  isempty, number_elements, enqueue, dequeue, first

feature specification

  isempty: BOOLEAN
    -- Is queue empty?

  number_elements: INTEGER
```

[16] Smalltalk is the most famous programming environment to emerge from the Xerox Palo Alto Research Center (PARC). Mesa and Cedar are examples of more recent environments produced by Xerox PARC.

```
  enqueue(x: XTYPE)
    -- Enqueue x.
  ensure
    number_elements = old number_elements + 1

  dequeue
    -- Delete first element in queue.
  require
    not isempty
  ensure
    number_elements = old number_elements - 1

  first: XTYPE
    -- Value of first element in queue.
  require
    not isempty

end interface -- class QUEUE
```

This is a class interface specification similar to a package specification in Ada. Since interfaces are not coded separately from implementations in Eiffel, the above is not even valid Eiffel code. This queue interface is actually the output of a tool supplied with the compiler. In Eiffel, one codes a class as one would code a package body in Ada. This tool, called Short, then generates an interface specification to provide programmers with the information to use that class while satisfying principles of information hiding and data abstraction.

However, Short does not support project management as do Ada packages specifications. A Short specification is only produced from a completely implemented class. Eiffel does provide a separate facility, the ability to defer the implementation of a class, that permits an Eiffel program to be developed similarly to the Ada process model[17]. But this facility is not integrated with Short.

As well as supporting inheritance, Eiffel provides generics. In fact, the sample queue specification is a generic containing the generic parameter XTYPE. Eiffel generics are not nearly as full featured as Ada generics. They are provided only to support those applications that are not quite as convenient to implement with inheritance.

Eiffel is purposely designed to allow exceptions to be reasoned about formally. Ada exceptions can be used to create programs even more difficult to reason about than those loaded with gotos. Yet they address the need to gracefully handle malfunctioning sensors and interrupts. Eiffel formally introduces preconditions and postconditions. These are seen in the "require" and "ensure" clauses of the above queue example. When a precondition or postcondition is not met, an exception is raised and control is returned to the calling procedure. Eiffel clearly restricts use of exceptions; exceptions are not allowed to propagate willy-nilly through an Eiffel program.

Eiffel demonstrates that support for a particular programming paradigm is not the only measurement criteria in evaluating programming languages. Efficiency, ease of reasoning about

[17] Ada package specifications allow a few skilled designers to outline the interfaces to the packages in a system before many others actually implement them. With Ada and Modula-2, the separation of specifications from implementation allows coders to proceed in relative isolation from one another.

programs, powerful abstraction mechanisms, and support for parallel processing might all be relevant. Eiffel integrates pure Object Oriented capabilities with powerful advances brought forth in Ada unrelated to OOD.

# 4. LARGER IMPLICATIONS

Software is inherently complex, and no foreseeable technology can be expected to remove that complexity[18]. Despite this lack of foreseeable technological breakthroughs, some ongoing research does deal with the conceptual essence of software. OOD attacks essential difficulties in software design. It also shows great promise in supporting software reuse. Finally, it has interesting implications on alternative software lifecycle models.

Various criteria can be used to evaluate a software design methodology. The methodology should result in a solution that mirrors the problem domain. Guidance should be provided on how to decompose a problem into smaller, "mind sized," problems. The notation in which the design is expressed should reflect this development and be readily comprehensible to readers. Finally, the methodology should be based on a rigorous theory.

OOD was developed to address certain gaps in structured programming. Top down design scatters related matters, particularly those involving data structures throughout the code. Although the program may have begun in terms of the problem domain, the final notation tends not to reflect this orientation in any easily visible manner. Top down design can encourage the development of tightly coupled procedures that depend for correct operation on the sequence in which they are called. The resulting loss of comprehensibility and lack of composability becomes particularly debilitating on large projects.

OOD is effectively a continuation of the structured programming revolution to meet the challenges of "programming in the large," the design of large systems over an extended period. It provides a method for decomposing a problem without falling into the traps of structured programming. Like top down design, OOD allows a solution to be specified in terms of the problem domain. The concepts from the problem domain, however, are objects, not functions, objects (data) being more stable than functions. Further, since decomposition is based on objects, components can be more easily combined to solve new problems. By allowing data and procedures in the resulting design to be grouped into modules or classes, the final notation should clearly reflect the design and the problem domain. Finally, the use of Abstract Data Types provides the needed formalism to clearly reason about Object Oriented Designs.

---

[18] This inherent complexity reflects the difficulties in scaling up over many levels of abstraction, the nonexistence of any adequate mathematical theory for analyzing a very large number of discrete states, the requirement that software conform to arbitrary interfaces dictated by the people and institutions it supports, the pressures on software to constantly change, and the difficulty in visualizing software architectures [Brooks 87].

## 4.1 Software Reuse

OOD seems to be the most promising design method available now. OOD promises to offer advantages particularly in encouraging software reuse. Reuse can include much more than code; "any information which a developer may need in the process of creating software [Freeman 87]" (e.g. specifications, plans, designs, code, and test cases) may be reused. Cataloging and retrieval schemes for this information can be a key problem in using reuse to develop software. In addition to technical concerns, reuse issues include economic, legal, and institutional concerns. Finally, reuse requires a certain mindset not necessarily common among many programmers. OOD, however, only addresses a limited range of these issues.

OOD promises to dramatically increase the reuse of software, thus reaping these benefits of lower cost and increased reliability. As a technological solution to some of the problems of the software crisis, reusability raises productivity. If a given functionality can be delivered by constructing a software system from existing components, rather than by developing new code, that functionality will be produced at a lower cost. Reuse has economic benefits since the cost of new development can be amortized over all future projects that reuse the products of the original development. Finally, reuse can improve the reliability of software products since often used code should be more reliable than code developed for one-time use only.

Adopting OOD requires a new mindset on the part of software developers. OOD encourages programmers to concentrate on the architecture of systems and to consider systems as composed of objects. Such an attitude should result in programmers being more willing to reuse objects without feeling any limitations on their creativity.

Several properties suggest that OOD can very successfully promote the reuse of code. Since objects provide a convenient packaging mechanism, reused code often need not be modified. If it does, programmers can modify objects by inheriting reused functionality from libraries and only modifying what they need in the new objects. The library itself can remain unchanged. OOD then supports the development of libraries of classes.

OOD researchers have developed techniques that address some of the cataloging and retrieval difficulties of reusability. Smalltalk environments include a tool, Browser, that is useful for quickly examining libraries of classes. When deciding whether or not to use a library component, a programmer needs a mechanism for determining what a component does, other than reading the entire source code. Well documented Ada package specifications and Modula-2 module definitions provide such a mechanism.

OOD's promise of software reusability has already been successful in a few cases. A taxonomy for reusable software components has been proposed [Booch87a]. Components with different time and

space characteristics (e.g. dynamic allocation of memory versus a fixed size at compile time) have been developed (stacks, lists, strings, queues, rings, sets, and trees) using the Object Oriented Design methodology. Although not all variations permitted by Booch's taxonomy are filled yet, the resulting complete set of components is now commercially available.

The NASA Goddard Space Flight Center (NASA/GSFC) has experimented with Ada and OOD in the Software Engineering Laboratory (SEL) in the last few years. They concluded that Ada needs to be taught with a design methodology; otherwise, the result is "Adatran" code, Ada code written in a Fortran style. As a consequence of experiments with OOD, the SEL produced two OOD systems[19] particularly oriented towards producing reusable components.

Interestingly, both of these successful uses of OOD to produce reusable components produced Ada packages. Noting that Ada does not fully support OOD, these results suggest that the reusability benefits of OOD at the least, can be achieved with only a partial implementation. The ability to experiment with an innovation on a limited basis is one characteristic that will lead to an innovation being rapidly adopted [Rogers 1983].

## 4.2 Life Cycle Support for OOD

OOD can be used in a project managed with the traditional waterfall life cycle[20] model. Basically, Object Oriented Design is a method for performing the design and coding phases of the waterfall model. In particular, OOD is strongly oriented toward the architectural design performed during preliminary design. Because of its close connections to certain programming languages, however, OOD also supports the detailed design (and coding) phases.

The most obvious restriction that OOD places on tools and methods occurs during the coding phase. One of greatest benefits OOD is likely to provide is lower maintenance costs resulting from

[19] One was coded in Fortran and the other in Ada. Successive projects tried to reuse this code. The Fortran project following did not have a higher percentage of reused code than is typical of NASA/GSFC projects. The Ada project following, however, was constructed with ninety percent reused code. The SEL concluded that this percentage is higher than can normally be expected. Nevertheless, the SEL does believe that OOD, as implemented in Ada, leads to more reusable code [Seidewitz 89].

[20] The waterfall model envisions software systems developing by moving in order through a sequence of phases. Although the exact phases vary from one model to another, a typical model might include requirements, preliminary design, detailed design, code and unit testing, integration testing, and system testing. DOD-STD-2167A, Defense System Software Development, defines the phases comprising the software development life cycle:

Software requirements analysis

Preliminary design

Detailed design

Coding and Computer Software Unit (CSU) testing

Computer Software Component (CSC) integration and testing

Computer Software Configuration Item (CSCI) testing

OOD's increased flexibility. Reducing maintenance costs that are 40 to 70 of the total cost for large systems can be significant ([Boehm 81] and [Pressman 87]). Unless the coding language reflects the Object Oriented nature of the design, much of the potential for reducing this cost will be lost.

OOD can also be used as a rapid prototyping tool during the requirements phase, particularly using the Smalltalk environment. By making a large library of predefined classes available, Smalltalk systems allow one to rapidly construct a system by linking already existing classes. This increased flexibility also encourages a style of exploratory programming needed for rapid prototyping.

The use of OOD as a design method, however, imposes restrictions on what methods can be used during requirements analysis. A method oriented more toward data structures[21] and less toward system functionality[22] can be expected to work better with OOD.

OOD promises to give broad support for software reuse, but the waterfall model does not fit well with reuse. No task performed under the waterfall model is explicitly oriented towards either using existing libraries or producing reusable components. Reuse requires conscious direction by management.

OOD attempts to build more flexible systems. The rigidity of the waterfall model hinders the continual adaptation of a design as its role becomes more precisely understood. Obviously, this rigidity of the waterfall model counters the flexibility sought by means of OOD. Changing a life cycle model can be very costly, but the modifications suggested for OOD fit into a natural evolution that seems to be occurring already.

OOD, then, can fit the conventional waterfall model, but alternative models may take fuller advantage of the increased reusability and flexibility that OOD promises. Various models have been suggested to improve the software life cycle. A spiral model of repeated builds has been proposed [Boehm 88]. In adapting the Constructive Cost Model (COCOMO) for Ada, Boehm has also proposed an Ada process model that fits comfortably with OOD. In this model a small team of designers begins by designing class interfaces and Ada package specifications. The team grows so as to implement these classes. The division between design and coding phases is less rigid in this model than in many waterfall models. In particular, many Critical Design Reviews (CDRs) are conducted, each concentrating on a particular class or subsystem.

---

[21] The two most well known data structure-oriented methods are probably Jackson System Development (JSD) ([Jackson 75] and [Jackson 83]) and Data Structured Systems Development, also known as the Warnier-Orr methodology [Pressman 87]. Grady Booch has found OOD to work well with JSD [Booch 87b].

[22] The most well-known requirements method oriented towards system functionality is undoubtably Structured Analysis (SA) [DeMarco 79]. One might expect SA to work less well with OOD; in fact, SA incorporates its own design methodology which is known as transform analysis. Practitioners, however, have built up a great deal of experience applying SA to a wide variety of systems under a wide variety of conditions. Some have even found it possible to apply SA with OOD [Ward 89].

The adoption of a new model based on rapid prototyping has also been suggested. By intertwining specification and implementation, systems users are better accommodated. Smalltalk itself grew up in an environment in which exploratory programming is encouraged. This history is strong evidence that OOD should fit well with these new paradigms[23] for software development.

[23] A good overview of these alternative life cycle models is provided by [Agresti 86].

# 5. SUMMARY

OOD is an exciting design methodology that is currently garnering a lot of attention. By providing a brief introduction to the technical features of OOD and the core elements of the methodology, this report has provided a sense of what the excitement is all about.

For some time now analysts have been grappling with the essential difficulties of building large software systems. An OOD methodology was presented in Section 2 that, while promising no miracle cure, can guide system developers toward more flexible and robust solutions. As was mentioned in Section 1, the research that led to this methodology grew out of concerns with issues not addressed by the structured programming revolution, most notably inadequate guidance on modular decomposition and a lack of flexibility in the resulting systems. Now that practitioners have seen these problems cause real systems to fail, a methodology developed to meet these problems, OOD, is becoming increasingly popular.

Ultimately, all software systems are expressions in some programming language. For a design methodology to be successful, the resulting solution needs to be clearly stated in the programming language. Section 3 evaluated several languages for their support for OOD. Some of these languages are fairly obscure, but the languages that are being selected by the marketplace these days show a strong OOD flavor. The results and scale for rating languages presented in Section 3 will help the reader choose a programming language for his system. They will also help those who have no choice better use the languages they have to support OOD.

Section 4 has shown that considerable benefits can be obtained from OOD, particularly in the area of reusability. OOD also fits well with the prototyping orientation of new lifecycles designed to overcome the rigidity of the traditional waterfall model. To obtain these benefits, Defense contractors face potentially increased costs of gearing up for OOD, resistance to the needed paradigm changes, and complications of applying OOD against mandated development standards. If recent history is any guide, however, OOD will continue to become ever more widely used and will play a large role in the future development of software engineering.

# 6. REFERENCES


[Abbott 83] Abbott, Russell J., "Program Design by Informal English Descriptions," *Communications of the ACM*, Volume 26, Number 11, November 1983.

[Abbott 86] Abbott, Russell J., *An Integrated Approach to Software Development*, John Wiley & Sons, 1986.

[Agresti 86] Agresti, William W., *New Paradigms for Software Development*, IEEE Computer Society, 1986.

[ANSI83] *Reference Manual for the ADA Programming Language*, ANSI/MIL-STD-1815A, United States Department of Defense, January 1983.

[Backus 78] Backus, John, "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs," *Communications of the ACM*, Volume 21, Number 8, August 1978.

[Boehm 81] Boehm, Barry W., *Software Engineering Economics*, Prentice-Hall, 1981.

[Boehm 88] Boehm, Barry W., "A Spiral Model of Software Development and Enhancement," *Computer*, May 1988.

[Booch 86] Booch, Grady, "Object-Oriented Development," *IEEE Transactions on Software Engineering*, Volume 12, Number 12, February 1986.

[Booch 87a] Booch, Grady, *Software Components with Ada*, Benjamin/Cummings Publishing Company, Inc., 1987.

[Booch 87b] Booch, Grady, *Software Engineering with Ada*, Benjamin/Cummings Publishing Company, Inc., first edition 1983, second edition 1987.

[Brooks 75] Brooks, Frederick P., Jr, *The Mythical Man-Month*, Addison-Wesley, 1975.

[Brooks 87] Brooks, Frederick P., Jr, "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, April 1987.

[CAMP 87] "Version Description Document for the Missile Software Parts of the Common Ada Missile Packages (CAMP) Project," McDonnell Douglas Astronautics Company, 30 October 1987.

[Cointe 87] Cointe, Pierre, "Metaclasses are First Class: The ObjVlisp Model," *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87) Conference Proceedings*, October 1987.

[Cox 86] Cox, Brad J., *Object Oriented Programming: An Evolutionary Approach*, Addison-Wesley Publishing Company, 1986.

[DeMarco 79] DeMarco, Tom, *Structured Analysis and System Specification*, Yourdon Press, 1979.

[DOD-STD-2167A] *Defense System Software Development*, DOD-STD-2167A, 29 February 1988.

[Freeman 87] Freeman, Peter, "Reusable Software Engineering: Concepts and Research Directions," *Tutorial: Software Reusability*, The Computer Society of the IEEE, 1987.

[Jackson 75] Jackson, Michael A., *Principles of Program Design*, Academic Press, 1975.

[Jackson 83] Jackson, Michael A., *System Development*, Prentice-Hall International, 1983.

[Lieberherr 89] Lieberherr, Karl J. and Holland, Ian M., "Assuring Good Style for Object-Oriented Programs," *IEEE Software*, September 1989.

[Meyer 86] Meyer, Bertrand, "Genericity versus Inheritance," *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86) Conference Proceedings*, September/October 1986.

[Meyer 88] Meyer, Bertrand, *Object-Oriented Software Construction*, Prentice Hall International Limited, 1988.

[Minsky 87] Minsky, Naftaly H. and Rozenshtein, David, "A Law-Based Approach to Object-Oriented Programming," *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87) Conference Proceedings*, October 1987.

[Parnas 72] Parnas, David L., "On the Criteria to be used in Decomposing Systems into Modules," *Communications of the ACM*, Volume 15, Number 12, December 1972.

[Perez 88] Perez, Eduardo, "Simulating Inheritance with Ada," *Ada Letters*, Volume 8, Number 5, September/October 1988.

[Pressman 87] Pressman, Roger S., *Software Engineering: A Practitioner's Approach*, Second edition, McGraw-Hill Book Company, 1987.

[Reynolds 87] Reynolds, Charles W., "On Implementing Generic Data Structures in Modula-2," *Journal of Pascal, Ada, & Modula-2*, Volume 6, Number 5, September/October 1987.

[Rogers 83] Rogers, Everett M., *Diffusion of Innovations*, Third edition, Free Press, 1983.

[Seidewitz 89] Seidewitz, E. and Stark, M., "Ada in the SEL: Experiences with Operational Ada Projects," *Second NASA Ada Users' Symposium*, November 1989.

[Stein 87] Stein, Lynn Andrea, "Delegation is Inheritance," *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87) Conference Proceedings*, October 1987.

[Stevens 89] Stevens, Al, "From C to C++," *Dr. Dobb's Jornal*, Winter 1989.

[Stroustrup 88] Stroustrup, Bjarne, "What is Object-Oriented Programming?" *IEEE Software*, May 1988.

[Ward 89] Ward, Paul T., "How to Integrate Object Orientation with Structured Analysis and Design," *IEEE Software*, March 1989.

[Wiener 85] Wiener, Richard S. and Sincovec, Richard F., "Two Approaches to Implementing Generic Data Structures in Modula-2," *ACM SIGPLAN Notices*, Volume 20, Number 6, June 1985.

[Wiener 89] Wiener, Richard S. and Pinson, Lewis J., "A Practical Example of Multiple Inheritance in C++," *SIGPLAN Notices*, Volume 24, Number 9, September 1989.

[Wolf 89] Wolf, Wayne, "A Practical Comparison of Two Object-Oriented Languages," *IEEE Software*, September 1989.

# Appendix A.

# GLOSSARY

**Abstract Data Type (ADT)** - A data type from which implementation details are abstracted. The properties of an ADT are defined formally, typically by a collection of axioms.

**Abstract State Machine** - An automaton with well defined states; a means of detecting the current state; and a mechanism for switching from one state to another.

**Ada** - A programming language mandated by the Department Of Defense (DoD) to be used for all mission critical software. Ada's design began in the 1970s.

**Ancestor** - A class that provides features inherited by another class. Contrast Descendant.

**C++** - An Object Oriented extension of the programming language C.

**Class** - 1. A type and its associated attributes and operations. Instances of a class are known as objects. 2. A language construct combining module and type aspects.

**Client** - A class that makes use of the services provided by a given class. Contrast Supplier.

**CLU** - A programming language developed at the Massachusetts Institute of Technology during the 1970s. CLU has many Object Oriented features, although it does not support inheritance.

**Constructor** - An operation that either creates an object or changes its state. Contrast Destructor, Iterator, and Selector.

**Data Abstraction** - the separation of the specification of a data structure or object from its implementation. Data abstraction allows an implementation to be changed without its use being affected.

**Descendant** - A class that inherits features of another class. Contrast Ancestor.

**Destructor** - An operation that destroys an object. Contrast Constructor, Iterator, and Selector.

**Dynamic Binding** - Also known as Late Binding. A language feature of Object Oriented languages in which any ambiguities in the meaning of an identifier are resolved at run time. For example, since the types of several polymorphic variables used as arguments to a given invocation of a procedure might change during program execution, that invocation must be bound to several different but identically named procedures during run time. See polymorphism and contrast static binding.

**Eiffel** - An Object Oriented language and environment developed in the 1980s.

**Flavors** - An Object Oriented extension of the programming language LISP.

**Garbage Collection** - the deallocation of space occupied by inaccessible data items or objects.

**Generics** - Modules in which certain parameters are instantiated at run time. For example, an Ada generic package might specify a stack such that the same source code can be used to instantiate a stack of integers and a stack of characters.

**Inheritance** - A feature of Object Oriented languages in which a class can be defined as an extension or specialization of another class. This feature is analogous to the ability to define subtypes in traditional languages.

**Information Hiding** - A principle of program design in which implementation details are not available to modules using a given module or client classes of a given class.

**Iterator** - An operation that permits all parts of an object to be read or updated. Contrast Constructor, Destructor, and Selector.

**Loops** - An Object Oriented extension of the programming language LISP.

**Message** - A request that an operation be performed on an object; terminology used in Smalltalk. See also Method.

**Method** - An operation; terminology used in Smalltalk. The class of an object must find the appropriate method to apply when a message is sent to that object.

**Modula-2** - A programming language introduced in the early 1980s by Niklaus Wirth, the designer of Pascal. Although simpler, Modula-2 shares many of the features of Ada.

**Module** - A collection of data types, constants, variables, procedures, and functions. Typically modules can be separately compiled.

**Multiple Inheritance** - A language feature of Object Oriented languages in which classes can inherit features from more than one class. In other words, a descendant class can have more than one direct ancestor.

**Object** - An entity that has a state, is characterized by the operations that can be performed on it, is denoted by a name, is an instance of a class, and can be viewed by its specification as well as its implementation.

**Object Oriented Design** - the construction of software systems as structured collections of Abstract Data Type implementations.

**Objective C** - An Object Oriented extension of the programming language C.

**Operation** - A means of accessing the state of an object. Operations are analogous to procedures in traditional programming languages. They can be constructors, destructors, iterators, or selectors.

**Overloading** - A language feature in which an identifier can have several alternative meanings at a given point in the program text. For example, several different procedures might share the same name; the compiler would typically resolve any ambiguities based on the number and type of arguments. Compare Polymorphism.

**Polymorphism** - A language feature in which a program entity, such as a variable or object, can refer to instances of different types or classes at run time. Since the operations of these classes may share the same name, the meaning of an identifier at a given point in the program text may change during program execution. Compare Overloading.

**Repeated Inheritance** - A special case of multiple inheritance in which a class has ancestors through more than one route. For example, if class A inherits properties from classes B and C, and classes B and C each inherit properties from D, the relationship between A and D is one of repeated inheritance.

**Selector** - An operation that returns information on an object's state. Contrast Constructor, Destructor, and Iterator.

**Simula** - A programming language introduced in 1967, sometimes called Simula-67. Simula introduced the class feature of Object Oriented languages.

**Smalltalk** - An Object Oriented language and environment developed at the Xerox Palo Alto Research Center in the 1970s.

**Static Binding** - Also known as Early Binding. A language feature in which any ambiguities in the meaning of an identifier are resolved at compile time. For example, the compiler might bind a given invocation of a procedure to one of several identically named procedures based on the number and type of arguments. See overloading and contrast dynamic binding.

**Strong Typing** - The enforcement of the type of a variable or the class of an object, which prevents variables of different types from being interchanged or, at most, allows them to be interchanged only in very restricted ways. For example, in a strongly typed language, it might be a syntax error to attempt integer arithmetic on pointers. Contrast Weak Typing.

**Supplier** - A class that provides services which other classes can use. Contrast Client.

**Weak Typing** - The nonenforcement of the type of a variable or the class of an object, which allows variables of different types to be interchanged and combined in many ways. For example, in a weakly typed language, one might be able to add integers to boolean variables or perform integer arithmetic on pointers. Contrast Strong Typing.

# Appendix B.

# ANNOTATED BIBLIOGRAPHY


168 Brooks, Frederick P., Jr; *THE MYTHICAL MAN-MONTH ESSAYS ON SOFTWARE ENGINEERING.* 206 p. Avail. from Addison-Wesley, Benjamin/Cummings Publ. Co., Inc., Jacob Way, Reading, MA 01867. Order No. ISBN 0-201-00650-2.

Key words:

An eminent computer expert, Brooks has written a collection of thought- provoking essays on the management of computer programming projects. These essays draw from his own experience as project manager for the IBM System/360 and for OS/360, its operating system.

In the essays, the author blends facts on software engineering with his own personal opinions and the opinions of others involved in building complex computer systems. He not only gives the reader the benefit of the lessons he has learned from the OS/360 experience, but he writes about them in an extremely readable and entertaining way.

Although formulated as separate essays, the book expresses a central argument. Brooks believes that large programming projects suffer management problems different in kind from small ones due to the division of labor. For this reason he feels that the critical need is for conceptual integrity of the product itself, and in essay form he explores both the difficulties of achieving this unity and the methods for achieving it.

2305 Parnas, David L.; "ON THE CRITERIA TO BE USED IN DECOMPOSING SYSTEMS INTO MODULES," In *Tutorial on Software Design Techniques.* Apr 1980. pp. 220-225. Avail. from IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854. Order No. EHO 161-0.

Key words:

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented, and both a conventional and unconventional decomposition are described. It is shown that the unconventional decomposition have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

2341 U.S. Dept. of Defense; *REFERENCE MANUAL FOR THE ADA PROGRAMMING LANGUAGE.* Report No. ANSI/MIL-STD-1815A. 330 p. Avail. from Naval Publications and Forms Center, 5801 Tabor Avenue, Philadelphia, PA 19120.

Key words:

This standard specifies the form and meaning of program units written in Ada*. Its purpose is to promote the portability of Ada programs to a variety of data processing systems. This standard specifies the form of a program unit written in Ada; the effect of translating and executing such a program unit; the manner in which program units may be combined to form Ada programs; the predefined program units that a conforming implementation must supply; the permissible variations within the standard, and the manner in which they must be specified; those violations of the standard that a conforming implementation is required to detect, and the effect of attempting to translate or execute a program unit containing such violations, and those violations of the standard that a conforming implementation is not required to detect. (*Ada is a trademark

of the U.S. Department of Defense).

2946 Boehm, Barry W.: *SOFTWARE ENGINEERING ECONOMICS.* 767 p. Avail. from Prentice-Hall, Rt. 59 at Brook Hill Drive, West Nyack, NY 10995. Order No. ISBN 0-13-822122-7.

Key words:

This textbook's objective is to provide a basis for a software engineering economics course. The book discusses, in detail, economic considerations for the development and maintenance of computer software. As background, the author provides two case studies involving the development of two new systems. Then, the goals of software engineering are described. The basic thrust of the text then is to present techniques, tools, and models for project planning, cost estimation, decision analysis, risk analysis, and other management perspectives. Some of the approaches and techniques discussed include the following: the Goal-Oriented Approach to Life-cycle Software, the Constructive Cost Model, the prototype approach, Rayleigh Distributions, Bayes' Formula, and the Value-of-Information Approach. The book ends with a chapter devoted to suggestions for improving productivity on software projects.

5326 Cox, Brad J.; "THE MESSAGE/OBJECT PROGRAMMING MODEL," In *Softfair Conference on Software Development Tools, Tech & Alt Proceedings (1983).* Jul 1983. pp. 51-60. Avail. from IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854. Order No. 478.

Key words:

This is a tutorial on the programming style used in Smalltalk 80, and a personal history of the reasoning that has led the author to pursue this style within conventional languages like C. It addresses the questions "What is message/object programming?", "How is it different from conventional programming?", and "What can be gained by adopting it?". (author)

5507 Freeman, Peter; "REUSABLE SOFTWARE ENGINEERING: CONCEPTS AND RESEARCH DIRECTIONS," In *Tutorial on Software Design Techniques.* Aug 1983. pp. 63-78. Avail. from IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854. Order No. 514.

Key words:

The objective of reusable software engineering is to enable the broad reuse of all types of information found in development situations. This paper defines classes of information to be reused, discusses the processes and conditions surrounding reuse, and suggests research tasks that will improve our ability to practice reuse. (author)

5665 Sodano, Nancy M.; Szulewski, Paul A.: "DESIGN METRICS AND ADA," In *Washington Ada Symposium (Mar. 1984).* Mar 1984. pp. 105-114. Avail. from ACM, Washington Chapter, P. O. Box 6228, Washington, DC 20015.

Key words:

This paper reports on work done in investigating the use of Ada as a Program Design Language (PDL), and the evaluation of Ada designs with a design metric. The first section provides background and describes the context for the work. The second section defines the Halstead metrics and discusses their application during the design phase. The third section discusses using Ada as a Program Design Language. The fourth section presents an example which illustrates the usefulness of the design metrics on the Ada PDL design medium. Finally, the conclusions of this work are presented. (author)

5676 Jackson, M.A.: *PRINCIPLES OF PROGRAM DESIGN.* 299 p. Avail. from Academic Press Inc., 111 Fifth Avenue, New York, NY 10003. Order No. ISBN 0-12-379050-6.

Key words:

This book describes how to design structured programs such that the resulting programs will be easy to understand, easy to maintain, free from logic errors, and structured like the problem. The methodology advocated is based on the principle that program structures should be based on data structures. This methodology has three steps. First, one should consider the data

structures, which will then be used to form a program structure. Second, one lists the executable operations needed to carry out the task. Third, one allocates each operation to a component of the program structure. The quality of the work done when performing these steps determines the quality of the programs produced. The methodology is illustrated by numerous example COBOL programs.

5682 Cox, Brad J.; "MESSAGE/OBJECT PROGRAMMING: AN EVOLUTIONARY CHANGE IN PROGRAMMING TECHNOLOGY," In *IEEE Software.* 1(1): Jan 1984. pp. 50-61.

Key words:

This article is a tutorial on the object-oriented programming style used in Smalltalk-80. The author discusses message/object programming, how it differs from conventional programming, and how it can be achieved through software evolution as opposed to revolution. The author concludes that Smalltalk's dynamically bound message/object paradigm solves several key problems that can prevent programmers from building highly malleable, reusable software.

5692 Sincovec, Richard F.; Wiener, Richard S.; "MODULAR SOFTWARE CONSTRUCTION AND OBJECT-ORIENTED DESIGN USING ADA," In *Journal of Pascal, Ada & Modula 2.* 3(2): Mar 1984. pp. 29-34.

Key words:

This paper describes a software development methodology which refers to as modular software construction and object-oriented design. This powerful and modern approach to software development has recently gained tremendous currency with the advent of software engineering languages such as Ada and Modula-2. In this paper focus is made on the use of Ada in conjunction with this methodology. (author)

5701 Sincovec, Richard F.; Wiener, Richard S.; "MODULA SOFTWARE CONSTRUCTION AND OBJECT-ORIENTED DESIGN USING MODULA-2," In *Journal of Pascal, Ada & Modula 2.* 3(3): May 1984. pp. 41-47.

Key words:

In the March/April 1984 issue of this journal discussed is object-oriented design using Ada (see "Modula Software Construction and Object-Oriented Design Using Ada"). In this article this theme is continued but focus shifts to Modula-2. The steps presented in the March/April article for performing object-oriented design are briefly summarized. This article illustrates the process of object-oriented design with a case study. The subject of the case study is a tic-tac-toe game, human vs. computer because its design is complex enough to warrant object-oriented design. (author)

5971 "AN OVERVIEW OF SIGNAL REPRESENTATIONS IN SIGNAL PROCESSING LANGUAGES," pp. 69-73. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-P002 608.

Key words:

This paper reviews three approaches to the representation of discrete-time signals as objects in programs. The first two representations, arrays and streams, are widely used in contemporary signal processing programming. The third representation was introduced in the recently-proposed Signal Representation Language (SRL). SRL signals are abstract objects whose properties are explicitly designed to reflect those of the represented signals. Arrays, streams, and SRL signal objects are discussed in the context of a set of signal representation criteria which are motivated by elementary observations about the mathematics of discrete-time signals. The emphasis in the paper is on the semantics of signal representation rather than on issues of time- or space-efficiency. (author)

6004 Horowitz, Ellis; *FUNDAMENTALS OF PROGRAMMING LANGUAGES.* 446 p. Avail. from Computer Science Press, Inc., 11 Taft Court, Dept. C1083, Rockville, MD 20850. Order No. ISBN 0-88175-004-2.

Key words:

This book takes a fundamentally different point of view from tracitional books on programming languages. The best possible way to study and understand today's programming languages is by focusing on a few essential concepts. These concepts form the outline for this book and include such topics as variables, expressions, statements, typing scope, procedures, data types, exception handling, and concurrency. by understanding what these concepts are and how they are realized in different programming languages, one arrives at a level of comprehension far greater than one gets by writing some programs in a few languages. Moreover, knowledge of these concepts provides a framework for understanding future language designs. Numerous examples from Ada, Pascal, LISP, and other programming languages are included. This book is a study of the complexities of programming languages. (author)

6035 Kehler, Thomas P.; Kunz, John C.; Williams, Michael D.; "APPLICATIONS DEVELOPMENT USING A HYBRID AI DEVELOPMENT SYSTEM," In *AI Magazine.* 5(3): Sep 1984. pp. 41-54.

Key words:

This article describes building applications programs in a hybrid Artificial Intelligence (AI) tool environment. Traditional AI systems developments have emphasized a single methodology, such as frames, rules, or logic programming, as a methodology that is natural, efficient, and uniform. The applications developed in this experiment suggest that naturalness, efficiency and flexibility are all increased by trading uniformity for the power that is provided by a small set of appropriate programming and representation tools. The tools used are based on five major AI methodologies: frame-based knowledge representation with inheritance, rule-based reasoning, LISP, interactive graphics, and active values. Object-oriented computing provides a principle for unifying these different methodologies within a single system. (author)

6043 Jamsa, Kris A.; "OBJECT ORIENTED DESIGN VS STRUCTURED DESIGN -- A STUDENT'S PERSPECTIVE," In *Software Engineering Notes (ACM SIGSOFT).* 9(1): Jan 1984. pp. 43-49.

Key words:

This paper discusses the advantages that structured design has over object-oriented design. The author favors structured design and presents a hierarchically organized collection of processes in order to emphasize the advantages of a graphic approach to design. The steps involved in object-oriented design, as well as, an illustration of Ada packages are presented. The author suggests that object oriented design places a burden on the designer at the interface stage due to its graphic shortcomings.

6362 Futatsugi, Kokichi; *HIERARCHICAL SOFTWARE DEVELOPMENT IN HISP.* pp. 151-174. Avail. from North Holland Publishing Company. Order No. ISSN 0167-50036.

Key words:

Software (specification, program, etc.) development is simply modeled as the incremental construction of a set of hierarchically structured cluster of operators. This paper presents the language HISP, which embodies this modeling. In this language, each software module (description unit) is the result of applying one of five module building operations to the already existing modules. This basic feature of the language makes it possible to write inherently hierarchical software. Using this property, many mechanisms for top-down software development are easily realized. Parameterized types, in particular, are available in the language by using these specific operations for module building. In this paper, the HISP language is introduced informally and the hierarchical software development in HISP is explained by use of simple examples. The present status of the HISP implementation is also sketched. (author)

6393 Weber, Herbert; "THE DISTRIBUTED DEVELOPMENT SYSTEM - A MONOLITHIC SOFTWARE DEVELOPMENT ENVIRONMENT," In *Software Engineering Notes (ACM SIGSOFT).* 9(5): Oct 1984. pp. 43-72.

Key words:

This paper contains a very coarse description of a new type of software development environment. It supports uniform specifications and renders itself on the basis of this uniform specification technique into a monolithic environment. The environment is composed of a number of support units. Some of them are meant to support users with the aid of expert knowledge maintained in those support units. All support units are interconnected in a tailored communication network that supports standard communication services. The paper presents work in progress. The described features of the Distributed Development System are, therefore, subject to changes. (author)

6471 Porubcansky, C.A.; *PROCEEDINGS OF THE 2ND AFSC AVIONICS STANDARDIZATION CONFERENCE. VOLUME 8. TUTORIAL: MIL-STD-1815 ADA HIGH ORDER LANGUAGE.* Report No. ASD(ENA)-TR-82-5031. 170 p. Nov 1982. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A142 783.

Key words:

This is volume 8 from a collection of nine volumes of unclassified papers to be distributed to the attendees of the Second Air Force Systems Command (AFSC) Avionics Standardization Conference at the Convention Center, Dayton, Ohio. The scope of the Conference includes the complete range of DOD approved embedded computer hardware/software and related interface standards as well as standard subsystems used within the Tri-service community and NATO. The theme of the conference is "Rational Standardization". Lessons learned as well as the pros and cons of standardization are highlighted. This volume is a tutorial that discusses the development history, design, and implementation of MIL-STD-1815 (the Ada programming language). The syntax and semantics of the language will be covered in overview fashion with emphasis on data typing and the use of Ada as an object-oriented design language. These view graphs are usable as the curriculum for an introductory class on the Ada language. (author)

6622 Novak, Gordon S., Jr.; "KNOWLEDGE-BASED PROGRAMMING USING ABSTRACT DATA TYPES," In *Proceedings of the National Conference on Artificial Intelligence.* Aug 1983. pp. 288-291. Sponsored by National Science Foundation, Washington, DC 20550. Grant/Contract No. SED-7912803. Sponsored by Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington VA 22209. Grant/Contract No. MDA-903-80-C-007.

Key words:

Features of the GLISP programming system that support knowledge-based programming are described. These include compile-time expansion of object-centered programs, interpretation of messages and operations relative to data type, inheritance of properties and behavior from multiple superclasses, type inference and propagation, conditional compilation, symbolic optimization of compiled code, instantiation of generic programs for particular data types, combination of partial algorithms from separate sources, knowledge-based inspection and editing of data, menu-driven interactive programming, and transportability between Lisp dialects and machines. GLISP is fully implemented for the major dialects of Lisp and is available over the ARPANET. (author)

6677 Boehm-Davis, D.A.; Ross, L.S.; *APPROACHES TO STRUCTURING THE SOFTWARE DEVELOPMENT PROCESS.* Report No. GEC/DIS/TR-84-B1V-1. 33 p. Sponsored by Office of Naval Research, 800 North Quincy St., Arlington, VA 22217. Grant/Contract No. N00014-83-C-0574. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A147 694.

Key words:

This research examined program design methodologies, which claim to improve the design process by providing strategies to programmers for structuring solutions to computer problems. In this experiment, professional programmers were provided with the specifications for each of three non-trivial problems and asked to produce pseudo-code for each specification according to the principles of a particular design methodology. The measures collected were the time to design and code, percent complete, and complexity, as measured by several metrics. These

data were used to develop profiles of the solutions produced by different methodologies and to develop comparisons between the methodologies. The results suggest that there are differences among the various methodologies. These differences are discussed in light of their impact on the comprehensibility, reliability, and maintainability of the programs produced. (author)

6726 Kornfeld, William A.; "EQUALITY FOR PROLOG." In *International Joint Conference on AI Inc., 1983 - Karlsruke, W. Germ.* Aug 1983. pp. 514-519.

Key words:

The language Prolog has been extended by allowing the inclusion of assertions about equality. When a unification of two terms that do not unify syntactically is attempted, an equality theorem may be used to prove the two terms equal. If it is possible to prove that the two terms are equal the unification succeeds with the variable bindings introduced by the equality proof. It is shown that this mechanism significantly improves the power of Prolog. Sophisticated data abstraction with all the advantages of object-oriented programming is available. Techniques for passing partially instantiated data are described that extends the "multiuse" capabilities of the language, improve the efficiency of some programs, and allow the implementation of arithmetic relations that are both general and efficient. The modifications to standard Prolog are simple and straightforward and in addition the computational overhead for the extra linguistic power is not significant. Equality theorems will probably play an important role in future logic programming systems. (author)

6876 Borger, Mark W.; "ADA SOFTWARE DESIGN ISSUES." In *Journal of Pascal, Ada & Modula 2.* 4(2): Mar 1985. pp. 7-14. Sponsored by Naval Ocean Systems Center, San Diego, CA 92152-5000. Grant/Contract No. N66001-82-C-0440.

Key words:

This article presents a discussion of specific experiences using Ada throughout the design of an Ada Programming Support Environment (APSE) software utility, namely the APSE Interactive Monitor (AIM). The AIM was designed using an object-oriented methodology with Ada as the Program Design Language (PDL). The intent of this article is to raise and discuss particular issues related to the use of the Ada language for both software design and development. It is not the intent to provide the reader with a tutorial on object-oriented design or the AIM program, nor to provide an introduction to the Ada language. (author)

7041 Sincovec, Richard F; Wiener, Richard S.; "TWO APPROACHES TO IMPLEMENTING GENERIC DATA STRUCTURES IN MODULA-2." In *ACM SIGPLAN Notices.* 20(6): Jun 1985. pp. 56-64.

Key words:

In this paper the authors present two approaches to implementing generic data structures in Modula-2. Both methods are illustrated with a generic search tree. The actual code as well as advantages and disadvantages are presented for both approaches.

7100 Maclennan, Bruce J.; *A SIMPLE SOFTWARE ENVIRONMENT BASED ON OBJECTS AND RELATIONS.* Report No. NPS52-85-005. 32 p. Apr 1985. Sponsored by Office of Naval Research, 800 North Quincy St., Arlington, VA 22217. Grant/Contract No. N00014-84-WR-24087. Sponsored by Office of Naval Research, 800 North Quincy St., Arlington, VA 22217. Grant/Contract No. N00014-85-WR-24057. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A155 704.

Key words:

This paper presents a simple programming system based on a clear separation of value-oriented programming and object-oriented programming. The value-oriented component is a conventional functional programming language. The object-oriented component is based on a model of objects and values connected by relations, and on production system-like rules that determine the alteration of these relations through time. It is shown that these few basic ideas

permit simple prototyping of a software development environment. (author)

7136 Clthoff, Walter; "AN OVERVIEW OF MODPASCAL." In *ACM SIGPLAN Notices*. 20(10): Oct 1985. pp. 60-71. Sponsored by Federal Ministry of Research and Technology, Federal Republic of Germany. Grant/Contract No. IT 8302363.

Key words:

In this paper the object oriented programming language ModPascal and its programming environment are introduced. ModPascal extends Standard Pascal by constructs that have shown usefulness in abstract data type theory such as module types, enrichments, instantiations and instantiated types. Also introduced is ModPascal editing, compiling, and execution using the ModPascal Programming System, which includes a multi-user data base of ModPascal objects.

7138 Rosenthal, Don; "ADDING META RULES TO OPS5: A PROPOSED EXTENSION," In *ACM SIGPLAN Notices*. 20(10): Oct 1985. pp. 79-86.

Key words:

In this paper, potential problems caused by the lack of explicit control constructs and the segregation of three memory areas in OPS5 were presented. Two solutions which allow such control constructs were presented. Both solutions implement meta-rules, the first by adding two constructs to the language, the second by allowing user-written conflict resolution strategies.

7251 Agusa, Kiyoshi; Ohno, Yutaka; Tarumi, Hiroyuki; "ACQUAINTANCE/INSTANCE VARIABLE MODEL FOR OBJECT-ORIENTED PROGRAMMING," In *COMPSAC 1985, Proceedings*. Oct 1985. pp. 69-73. Avail. from IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720. Order No. 0730-3157/85/0000/0069.

Key words:

This paper proposes a new model of object, called acquaintance/instance variable model. It reduces the complexity of object oriented representation of systems. This model gives a clear definition of an object's state, and definitions of effect and dependence between objects. The authors also give the basic concept of class-base. Class-base is a kind of database, which collects character descriptions of all classes programmers can use. A character description is based on the acquaintance/instance variable model. It describes internal and external features of a class. Internal features are related to the state of an object, whereas external features are related to other co-operative objects. With a Class-base, one can easily find classes and messages. (author)

7335 Agha, Bul Abdulnabi; *ACTORS: A MODEL OF CONCURRENT COMPUTATION IN DISTRIBUTED SYSTEMS*. Report No. 844. 198 p. Jun 1985. Sponsored by Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington VA 22209. Grant/Contract No. N00014-80-C-0505. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A157 917.

Key words:

A foundational model of concurrency is developed in this thesis. It examines issues in the design of parallel systems and shows why the actor model is suitable for exploring large-scale parallelism. Concurrency in actors is constrained only by the availability of hardware resources and by the logical dependence inherent in the computation. Unlike dataflow and functional programming, however, actors are dynamically reconfigurable and can model shared resources with changing local state. Concurrency is spawned in actors using asynchronous message-passing pipelining, and the dynamic creation of actors. The author defines an abstract actor machine and provides a minimal programming language for it. A more expressive language, which includes higher level constructs such as delayed and eager evaluation, can be defined in terms of the primitives. Examples are given to illustrate the ease with which concurrent data and control structures can be programmed. This thesis deals with some central issues in distributed computing. Specifically, problems of divergence and deadlock are addressed.

(author)

7465 Abbott, Russell J.; *AN INTEGRATED APPROACH TO SOFTWARE DEVELOPMENT.* 334 p. Avail. from John Wiley & Sons, Inc., 1 Wiley Drive, Attn: Order Dept., Summerset, NJ 08873. Order No. ISBN 0-471-82646-4.

Key words:

This book is intended as a text in software engineering courses and as a day-to-day working reference for practicing software engineers. It presents an integrated framework for software development that captures technical information needed to successfully develop and maintain a software system. This framework is presented in terms of a rationale for and outline of certain documents produced over the course of the life cycle. These documents include requirements documents, specification documents, and design documents. In addition, an appendix presents an easy-to-understand specification methodology that combines ideas from the predicate calculus and relational database design.

7554 Pitt, D. H.; Schuman, S. A.; *FORMAL TECHNIQUES FOR SPECIFICATION AND VALIDATION OF TACTICAL SYSTEMS.* Report No. CADD-8606-0203. 221 p. Jun 1987. Sponsored by Army Communication and Electronics Command (CECOM), Ft. Monmouth, NJ 07703. Grant/Contract No. DAAK80-81-C-0072. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A171 671.

Key words:

This document contains three appendices. The first appendix, "Object-Oriented Subsystem Specification," introduces a rigorous, mathematically based notation for supporting the earliest phases of the software design process. The second appendix, "An Experiment with an Approach to Formal Specifications," describes an experiment involving a new approach to system specifications. The process of developing a formal specification from an informally specified distributed information system concept forms the basis of the experiment. The last appendix, "Papers on Z," presents a concise summary of the mathematical sublanguage of the specification notation Z. (author)

7558 Agha, Gul; Hewitt, Carl; *CONCURRENT PROGRAMMING USING ACTORS: EXPLOITING LARGE-SCALE PARALLELISM.* Report No. AI 865. 21 p. Oct 1985. Sponsored by Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington VA 22209. Grant/Contract No. N0014-80-C-0505. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A162 422.

Key words:

The authors argue that the ability to model shared objects with changing local states, dynamic reconfigurability, and inherent parallelism are desirable properties of any model of concurrency. The actor model addresses these issues in a uniform framework. This paper briefly describes the concurrent programming language Act3 and the principles that have guided its development. Act3 advances the state of the art in programming languages by combining the advantages of object-oriented programming with those of functional programming. The authors also discuss considerations relevant to the large-scale parallelism in the context of open systems, and define an abstract model which establishes the equivalence of systems defined by actor programs. (author)

7593 Meyer, Bertrand; "EIFFEL: PROGRAMMING FOR REUSABILITY AND EXTENDABILITY," In *ACM SIGPLAN Notices.* 22(2): Feb 1987. pp. 85-94.

Key words:

Eiffel is a language and environment intended for the design and implementation of quality software in production environments. The language is based on the principles of object-oriented design, augmented by features enhancing correctness, extendibility and efficiency; the environment includes a basic class library and tools for such tasks as automatic configuration management, documentation and debugging. Beyond the language and environment aspect,

Eiffel promotes a method of software construction by combination of reusable and flexible modules. The present note is a general introduction to Eiffel. More detailed information is available. (author)

7625 Staff Author; *JOINT PROGRAM ON RAPID PROTOTYPING. RAPIER (RAPID PROTOTYPING TO INVESTIGATE END-USER REQUIREMENTS).* 297 p. Mar 1986. Sponsored by Office of Naval Research, 800 North Quincy St., Arlington, VA 22217. Grant/Contract No. N00014-85-C-0666. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A166 353.

Key words:

This report presents the results of work performed between July 1, 1985 and January 31, 1986 with the Office of Naval Research and Honeywell Computer Sciences Center. These results, and results obtained in the next several years, will be applied by the RAPIER (Rapid Prototyping to Investigate End-user Requirements) project in developing a software engineering environment to support prototyping for investigating end-user requirements. The environment supports a prototyping methodology, by which is meant a collection of techniques, a prescribed order for applying the techniques, reasons for the techniques, and their order of application. The RAPIER methodology will eventually contain techniques for each phase in the prototyping life cycle and for the transitions between phases. The RAPIER environment will contain software tools that support, encourage, and/or enforce these procedures and techniques. The RAPIER team develops new technology only when there is none available in either the commercial or the research marketplace. RAPIER is supported in part by the Department of Defense STARS Initiative's Application Area whose main thrust is software reusability. (author)

7631 Osterweil, Leon J.; *SOFTWARE PROCESS INTERPRETATION AND SOFTWARE ENVIRONMENTS.* Report No. DOE/ER/13283-5. 58 p. Apr 1986. Sponsored by National Science Foundation, Washington, DC 20550. Grant/Contract No. DCR-8403341. Sponsored by Department of Energy. Grant/Contract No. DE-FG02-84ER13283. Avail. from National Technical Information Service 5285 Port Royal Rd, Springfield, VA 22161. Order No. DE86010881.

Key words:

This paper suggests that a reasonable focus of software engineering is the notion of a "process-object"--namely an object which has been created by a development process, and which is itself a process. It then follows that the essence of software engineering is the study of effective ways of developing process-objects and of maintaining their effectiveness in the face of the need to make a wide variety of changes. These changes might entail alteration of the products produced by the process-object or alteration of the process-object itself. The main features of the insights and suggestions presented here revolve around the notion that process-objects must be defined in a precise, powerful, and rigorous formalism, and that once this has been done, the key activities of development, evaluation, and maintenance of both process-objects themselves, and their constituent parts alike, can and should be specified and implemented algorithmically. The suggested focus on process-objects draws a much-needed sharp line between software product development, evaluation and maintenance and software process development, evaluation and maintenance. This serves to improve one's understanding of both and to help to better understand the connections between such issues as maintenance, evaluation, reuse, and modularity. (author)

7642 Meyer, Bertrand; "REUSABILITY: THE CASE FOR OBJECT-ORIENTED DESIGN," In *IEEE Software.* 4(2): Mar 1987. pp. 50-64.

Key words:

Why isn't software more like hardware? Why must every new development start from scratch? This article addresses a fundamental goal of software engineering, reusability, and a companion requirement, extendibility (the ease with which software can be modified to reflect changes in specifications). The author's main thesis is that object-oriented design is the most promising

technique now known for attaining the goals of extendibility and reusability.

7643 Brooks, Frederick P., Jr; "NO SILVER BULLET: ESSENCE AND ACCIDENTS OF SOFTWARE ENGINEERING," In *Computer*. 20(4): Apr 1987. pp. 10-19.

Key words:

In this article, the author analyzes the nature of software engineering and assesses the technical developments that promise improvements in productivity, reliability, and simplicity. The author examines the inherent properties of modern software systems (complexity, conformity, changeability, and invisibility) and the promises and limitations of current software engineering research developments (Ada, object-oriented programming, artificial intelligence, expert systems, and graphical programming).

7720 Bradshaw, Susan M.; Byrne, William E.; Cronin, Neil A.; McDevitt, David E.; ***STRUCTURED HIERARCHICAL ADA PRESENTATION USING PICTOGRAPHS (SHARP) DEFINITION, APPLICATION AND AUTOMATION.*** Report No. ESD-TR-86-283. 348 p. Sep 1986. Sponsored by Air Force Electronic Systems Division (AFSC), Hanscom AFB, MA 01731. Grant/Contract No. F19628-84-D-0011. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A176 990.

Key words:

This paper presents a methodology for representing a large and complex computer program using graphics and Ada-based annotated pseudo code. It describes the application of the graphical representation, referred to as Structured Hierarchical Ada Representation using Pictographs (SHARP), in the design and test of computer programs, and presents a concept of operation for generating the graphics in a computer aided manner. The resulting tool is considered important, since design and test costs account for over 60 percent of software development costs. The tool also applies to software maintenance, which typically exceeds the original development cost by more than 50 percent. (author)

7728 Baker, Louis; "ADA AND AI JOIN FORCES," In *AI Expert*. 2(4): Apr 1987. pp. 38-43.

Key words:

While LISP is thought to be the language of choice for the development of artificial intelligence (AI) systems, there are reasons for contemplating the use of a general-purpose procedural programming language such as Ada for the production version of an AI system. For example, algorithms can be expressed in Ada and readily translated to other popular languages. The article first covers the requisite data structures and their implementation. A unification algorithm taken from a backward-chaining expert system is used to illustrate appropriate coding techniques. The article also briefly reviews how forward-chaining systems, augmented transition networks, frames, and object-oriented programming fit into implementation by general-purpose languages and by Ada. (author)

7739 Beloff, Bruno; Harland, David M.; "A PERSISTENT OBJECT STORE WITH AN INTEGRATED GARBAGE COLLECTOR," In *ACM SIGPLAN Notices*. 22(4): Apr 1987. pp. 70-79.

Key words:

This paper describes OBJEKT, a single-level persistent storage system designed for the REKURSIV architecture. It will be shown that OBJEKT can be microcoded to implement "objects" efficiently, and that data integrity can be guaranteed by provision of an object oriented instruction set. Particular attention will be paid to its facilities for type and range checking, to its object paging strategy and to ways to enancing parallelism during garbage collection. (author)

7740 Methfessel, Rand; "IMPLEMENTING AN ACCESS AND OBJECT ORIENTED PARADIGM IN A LANGUAGE THAT SUPPORTS NEITHER," In *ACM SIGPLAN Notices*. 22(4): Apr 1987. pp. 83-93.

Key words:

This article outlines some experiences in implementing an object oriented and access oriented paradigm in "C". These experiences originated in a graphics workstation development project which required graphics elements, which could appear in multiple windows, to respond immediately to changes in values in a user defined database. The number and type of graphics requiring updates, based on a change to a given database value, varied dynamically depending on which particular displays an operator had active at any given time. An object oriented paradigm has the values of variables change as a side effect of an object processing a message sent to it. The access oriented paradigm has a message sent as a side effect of an object variable changing. (author)

7763 Lovejoy, Alan; "EXTENSIONS TO MODULA-2." In *Journal of Pascal, Ada & Modula 2.* 6(2): Apr 1987. pp. 20-44.

Key words:

This article first briefly states some principles of language design. It then proposes some extensions to Modula-2. Each extension is illustrated by means of an example. These extensions are meant to correct some defects and limitations in current implementations, support functional and object-oriented programming styles, and give the language user greater control and power.

7809 Carey, Michael J.; Dewitt, David J.; Frank, Daniel; Graefe, Goetz; Muralikrishna, M.; Richardson, Joel E.; Shekita, Eugene J.; *THE ARCHITECTURE OF THE EXODUS EXTENSIBLE DBMS: A PRELIMINARY REPORT.* Report No. CONF-8609148-1. 36 p. May 1986. Sponsored by National Science Foundation, Washington, DC 20550. Grant/Contract No. MCS82-01870. Sponsored by National Science Foundation, Washington, DC 20550. Grant/Contract No. DCR-8402818. Sponsored by Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington VA 22209. Grant/Contract No. N00014-85-K-0788. Sponsored by Department of Energy. Grant/Contract No. DE-AC02-81ER10920. Avail. from National Technical Information Service 5285 Port Royal Rd, Springfield, VA 22161. Order No. DE86015438.

Key words:

With non-traditional application areas such as engineering design, image/voice data management, scientific/statistical applications, and artificial intelligence systems all clamoring for ways to store and efficiently process larger and larger volumes of data, it is clear that traditional database technology has been pushed to its limits. It also seems clear that no single database system will be capable of simultaneously meeting the functionality and performance requirements of such a diverse set of applications. This paper describes the preliminary design of an Extensible Object-oriented Database System (EXODUS), an extensible database system that will facilitate the fast development of high-performance, application-specific database systems. EXODUS provides certain kernel facilities, including a versatile storage manager and a type manager. In addition, it provides an architectural framework for building application-specific database systems, tools to partially automate the generation of such systems, and libraries of software components (e.g., access methods) that are likely to be useful for many application domains. (author)

7864 Boudreaux, J. C.; *OED: OBJECT-ORIENTED EDITOR.* Report No. NBSIR 87-3530. 17 p. Mar 1987. Avail. from National Technical Information Service 5285 Port Royal Rd, Springfield, VA 22161. Order No. PB87-173910.

Key words:

In this paper, the author describes an object-oriented editor, called OED, which is defined using the FranzLISP programming language. Though editors are usually associated with sets of functions to manipulate text-files, the author uses the term to characterize a family of LISP

functions which create and modify formal representations of objects in AMPLE/Core. (author)

7882 Braaten, Alan J.: *A GRAPHICS ENVIRONMENT SUPPORTING THE RAPID PROTOTYPING OF PICTORIAL COCKPIT DISPLAYS.* Report No. AFIT/GCS/MA/86D-1. 168 p. Dec 1986. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A178 636.

Key words:

Attention was focused on the interactive construction of pictorial type cockpit displays from libraries of cockpit displays and symbology. Implementation was based on an object-oriented programming paradigm. This approach provided a natural and consistent means of mapping abstract design specifications into functional software. Implementation was supported by an object-oriented extension to the 'C' programming language. Although this investigation addressed a specific application, the resulting graphic environment is applicable to other areas requiring the rapid prototyping of pictorial displays. (author)

7885 "PROCEEDINGS OF THE 5TH ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY HELD IN ARLINGTON, VIRGINIA ON MARCH 16-19, 1987," 510 p. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A178 690.

Key words:

The contents of this Ada Technology Proceedings include papers on the following topics: Reusability; Ada programming support environments; Applications; Language issues; Guidelines and standards; Commercial Ada users working group; Interoperability; Compilers; Hardware architectures; Project management; Methodologies; NASA application; Ada education and training; Metrics; DoD applications; Technology research; Portability; Performance issues; Distributed issues. (author)

8166 D'Ippolito, Richard; Lee, Kenneth; Plinta, Charles; Rissman, Michael; Van Scoy, Roger; *PROTOTYPE REAL-TIME MONITOR: DESIGN.* Report No. CMU/SEI-87-TR-38. 53 p. Nov 1987. Sponsored by SEI-Joint Program Office, Hanscom Air Force Base, Hanscom, MA 01731. Grant/Contract No. F1962885C0003. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A188 931.

Key words:

This report describes the software design used to implement the prototype real-time monitor (RTM) requirements. The prototype RTM described in this report was built to address two specific technical questions raised by the Ada Simulator Validation Program (AVSP) contractors: 1. How can user tools find, access and display data hidden in the bodies of Ada applications? 2. How can user tools be layered on top of Ada applications? The design is presented at three levels: system level, object level, and package architecture level. The report concludes with a discussion of the key implementation obstacles that had to be overcome to develop a working prototype: determining system addresses, communicating with an executing application, accessing application memory, converting data into human readable form, and distributed CPU architectures. (author)

8167 Van Scoy, Roger; *PROTOTYPE REAL-TIME MONITOR: ADA CODE.* Report No. CMU/SEI-87-TR-39. 180 p. Nov 1987. Sponsored by SEI-Joint Program Office, Hanscom Air Force Base, Hanscom, MA 01731. Grant/Contract No. F1962885C0003. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A191 095.

Key words:

This report presents the code that implements the prototype real-time monitor (RTM). In addition, the documentation in the package specifications and bodies forms the implementation description of the RTM. The prototype RTM described in this report was built to address two specific technical questions raised by the Ada Simulator Validation Program (AVSP) contractors:

1. How can user tools find, access and display data hidden in the bodies of Ada applications?
2. How can user tools be layered on top of Ada applications? (author)

8169 D'Ippolito, Richard; Lee, Kenneth; Plinta, Charles; Rissman, Michael S.; Van Scoy, Roger; *AN OOD PARADIGM FOR FLIGHT SIMULATORS.* Report No. CMU/SEI-87-TR-43. 101 p. Dec 1987. Sponsored by SEI-Joint Program Office, Hanscom Air Force Base, Hanscom, MA 01731. Grant/Contract No. F1962885C0003. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145.

Key words:

This report presents a paradigm for object-oriented implementations of flight simulators. It is a result of work on the Ada Simulator Validation Program (ASVP) carried out by members of the technical staff at the Software Engineering Institute (SEI). (author)

8220 Pressman, Roger S.; *SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH (SECOND EDITION).* 586 p. Avail. from McGraw-Hill Book Company, Princeton Road, Highstown, NJ 08520. Order No. ISBN 0-07-050783-X.

Key words:

This book's coverage of the software engineering process includes: planning and estimation techniques; analysis of the computer based system and the software element; design; coding; testing and quality assurance; and maintenance. Rather than maintaining a strict life cycle view, this second edition presents generic activities that are performed regardless of the software engineering paradigm that has been chosen. It offers a completely revised chapter on software testing and scheduling techniques; providing guidelines for cost/schedule estimation. This revision also features a new chapter on object-oriented design, real-time design, software test case design techniques, and software quality assurance. This coverage reflects new software engineering methods that are rapidly gaining acceptance in the industry. A well-designed learning tool; this edition contains many new problems, examples, and case studies oriented toward engineering/scientific systems and real-time applications. (author)

8223 Demarco, Tom; *STRUCTURED ANALYSIS AND SYSTEM SPECIFICATION.* 365 p. Avail. from Yourdon Press, 1133 Avenue of the Americas, New York, NY 10036-6748. Order No. ISBN 0-13-854380-1.

Key words:

This book is about Structured Analysis, and Structured Analysis is primarily concerned with a new kind of Functional Specification, the Structured Specification. Structured Analysis and System Specification describes an orderly approach to structured analysis and system design. The topics covered range from the basic concepts of system analysis, to the problems with system modeling. Functional decomposition, data flow diagrams, data dictionaries, process specifications, logical and physical models of a system are described in great detail, in a very readable manner. Explicit examples are given to walk the reader through the various stages necessary to perform structured analysis and specifications of a system in a logical manner. (author)

8225 Booch, Grady; *SOFTWARE ENGINEERING WITH ADA (SECOND EDITION).* 603 p. Avail. from Addison-Wesley, Benjamin/Cummings Publ. Co., Inc., Jacob Way, Reading, MA 01867. Order No. ISBN 0-8053-0604-8.

Key words:

This book has been written to satisfy the following three specific goals: to provide an intensive study of Ada's features; to motivate and give examples of good Ada design and programming style; to introduce an object oriented development method that exploits the power of Ada and, in addition, helps manage the complexity of large software systems. The book not only describes the details of Ada programming, but also suggests ways in which to best apply the features of the language in the creation of software systems. Software Engineering with Ada serves as a complete Ada reference that is appropriate for both the programmer who wishes to create Ada

systems and the manager who needs to understand how to apply this powerful tool. The book presumes an understanding of the basic principles of programming. Topics covered are: increased emphasis on Ada's syntax and semantics, detailed discussions on the object oriented development method, and updated historical information. (author)

8233 Blaha, Michael R.; Premerlani, William J.; Rumbaugh, James E.; "RELATIONAL DATABASE DESIGN USING AN OBJECT-ORIENTED METHODOLOGY," In *Communications of the ACM.* 31(4): Apr 1988. pp. 414-427.

Key words:

This article describes the effectiveness of the Object Modeling Technique (OMT) for approaching the design of relational databases. A comprehensive explanation of OMT is included, along with two applications showing the semantic improvement of OMT over other approaches for designing relational data base management systems. The design technique and methodology employed have been used for several years at General Electric, and the methodology is intuitive, expressive, and extensible. (author)

8335 Klahr, Philip; McArthur, David; Narain, Sanjai; "SWIRL: AN OBJECT-ORIENTED AIR BATTLE SIMULATOR," In *Proceedings of the National Conference on Artificial Intelligence, Aug. 18-20, 1982.* Aug 1982. pp. 331-334.

Key words:

The authors describe a program called SWIRL designed for simulating military air battles between offensive and defensive forces. SWIRL is written in an object-oriented language (ROSS) where the knowledge base consists of a set of objects and their associated behaviors. The authors discuss some of the problems they encountered in designing SWIRL and present their approaches to solving them. (author)

8403 Stroustrup, Bjarne; "WHAT IS OBJECT-ORIENTED PROGRAMMING?," In *IEEE Software.* 5(3): May 1988. pp. 10-20.

Key words:

This article presents the author's view of what object-oriented means in the context of a general-purpose programming language. Examples in C++ are presented, partly to introduce C++ and partly because C++ is one of the few languages that supports data abstraction, object-oriented programming, and traditional programming techniques. Issues of concurrency and hardware support for specific, higher level language constructs are not included. (author)

8436 "OOPSLA '87 CONFERENCE PROCEEDINGS," 636 p. Avail. from ACM Order Department, P. O. Box 64145, Baltimore, MD. 21264. Order No. ISBN 0-89791-247-0.

Key words:

This is an addendum to the proceeding of the OOPSLA '87 conference on object-oriented programming held in Orlando, Florida, October, 1987. It contains reports on five workshops and six panel discussions, in addition to the text of the keynote address and the banquet speech. These reports were written after the conference by organizers and attendees of these sessions in an attempt to capture the content, and some of the spirit, of these less formal technical exchanges. The reports have been organized into three areas of concern - roughly corresponding to design, implementation, and product development - and briefly summarized. (author)

8437 Diederich, Jim; Milton, Jack; "AN OBJECT-ORIENTED DESIGN SYSTEM SHELL," In *OOPSLA 1987 Proceedings.* Oct 1987. pp. 61-77.

Key words:

The authors present a design system shell which can be used to experiment with principles of design and be used as a design tool where complex layers of information need to be specified about objects, such as in database design. The shell can be tailored to a variety of application

areas. It is object-oriented in its implementation and structure. Objects and messages are used as the specification language. The basic ingredients of a rule-based production system are provided, with rules treated as objects and defined independently of the classes to which they are applied. (author)

8482 Boehm, Barry W.; "A SPIRAL MODEL OF SOFTWARE DEVELOPMENT AND ENHANCEMENT," In *Computer*. 21(5): May 1988. In *Tutorial: Software Engineering Project Management*. Jan 1988. pp. 61-72.

Key words:

This article opens with a short description of software process models and the issues they address. Subsequent sections outline the process steps involved in the spiral model; illustrate the application of the spiral model to a software project, using the TRW Software Productivity Project as an example; summarize the primary advantages and implications involved in using the spiral model and the primary difficulties in using it at its current incomplete level of elaboration; and present resulting conclusions. (author)

8515 Agusa, Kiyoshi; Ohno, Yutaka; Tarumi, Hiroyuki; "A PROGRAMMING ENVIRONMENT SUPPORTING REUSE OF OBJECT-ORIENTED SOFTWARE," In *10th International Conference on Software Engineering: April 11-15, 1988*. Apr 1988. pp. 265-273.

Key words:

The authors have developed a programming environment for object-oriented programming. This environment supports reuse of classes, especially retrieval of them with an expert system. The user can find classes and methods by describing the features of objects and operations according to an object model proposed by the authors. The target programming language is MOMO, which is developed by the authors to implement the object model. This paper mainly focuses on the retrieval part of the environment. (author)

8577 Meyer, Bertrand; "EIFFEL: A LANGUAGE AND ENVIRONMENT FOR SOFTWARE ENGINEERING," In *Journal of Systems and Software*. 8(3): Jun 1988. pp. 199-246.

Key words:

The Eiffel language and environment address the problem of building quality software in practical development environments. Two software quality factors were deemed essential in the design of the language: reusability and reliability. They led to the following choices: language features that support the underlying bottom-up software design methodology; modular structures based on the object-oriented approach, with support for both generic parameters and multiple inheritance (including a new extension, repeated inheritance); automatic storage management; highly dynamic execution model; support for polymorphism and dynamic binding; fully static typing; information hiding facilities; assertions and invariants that may be monitored at run-time. The Eiffel programming environment, using C as an intermediate language, supports separate compilation of classes and achieves a good run-time performance in both space and time. The environment takes care of automatically recompiling classes as needed after a change, ensuring that only up-to-date versions of classes are used, but avoiding unnecessary recompilations. A set of tools is provided to support the development of sizable software systems. An important part of the environment is the library of reusable classes. Significant extracts of this library are given in the appendix to this article, providing a set of model reusable software components, carefully designed for robustness and extendibility. (author)

8616 Agresti, William W.; *TUTORIAL: NEW PARADIGMS FOR SOFTWARE DEVELOPMENT*. 304 p. Jan 1986. Avail. from IEEE Computer Society, PO Box 80452, Worldwide Post Center, Los Angeles, CA 90080. Order No. ISBN 0-8186-0707-6.

Key words:

Designed for computer professionals who are interested in the process of software development, this tutorial shows the assumptions and limitations of the life-cycle (waterfall) model and explains when the model is appropriate and when it is not. Explains the new paradigms (prototyping,

operational specification, transformational implementation) and shows how they interrelate to support process improvement. Discusses the transition from the life-cycle model to a more flexible development process that accommodates these newer paradigms. (author)

8652 Danforth, Scott; Tomlinson, Chris; "TYPE THEORIES AND OBJECT-ORIENTED PROGRAMMING," In *ACM Computing Surveys.* 20(1): Mar 1988. pp. 29-72.

Key words:

Object-oriented programming is becoming a popular approach to the construction of complex software systems. Benefits of object orientation include support for modular design, code sharing, and extensibility. In order to make the most of these advantages, a type theory for objects and their interactions should be developed to aid checking and controlled derivation of programs and to support early binding of code bodies for efficiency. As a step in this direction, this paper surveys a number of existing type theories and examines the manner and extent to which these theories are able to represent the ideas found in object-oriented programming. Of primary interest are the models provided by type theories for abstract data types and inheritance, and the major portion of this paper is devoted to these topics. Code fragments illustrative of the various approaches are provided and discussed. The introduction provides an overview of object-oriented programming and types in programming languages; the summary provides a comparative evaluation of the reviewed typing systems, along with suggestions for future work. (author)

8662 Ramamoorthy, C. V.; Sheu, Phillip C.; "OBJECT-ORIENTED SYSTEMS," In *IEEE Expert.* 3(3): Sep 1988. pp. 9-15.

Key words:

Object-based systems provide such desirable features as data abstraction, program modularity, and inherent concurrency. The authors investigate the impact of object-based computation on databases and expert systems, and demonstrate object-based programming with a simple automatic factory example based on concepts of object, message, and class. The authors review the essence of object-oriented systems from a user's point of view, discussing problems that need to be resolved. In particular, the authors emphasize the need for object management systems, software engineering tools, and better architectural support. (author)

8726 "OOPSLA '88 CONFERENCE PROCEEDINGS," In *ACM SIGPLAN Notices.* 23(11): Nov 1988. Report No. 548881. 400 p. Avail. from ACM Order Department, P. O. Box 64145, Baltimore, MD. 21264. Order No. ISBN 0-89791-284-5.

Key words:

These proceedings contain papers presented at the Object-Oriented Programming Systems, Languages and Applications conference held on September 25-30, 1988. Such topics as implementation, user interfaces, extending Smalltalk, databases, tools and environments, applications, theory, concurrency and parallelism, and design were treated.

8773 Cesar, Edison M., Jr.; Ellis, John W., Jr.; Giarla, William; Klahr, Philip; Narain, Sanjai; Turner, Scott R.; *TWIRL: TACTICAL WARFARE IN THE ROSS LANGUAGE.* Report No. RAND/R-3158-AF. 59 p. Oct 1984. Sponsored by Air Force Research, Development and Acquisition, Hq Air Force, Washington, DC, 20330. Grant/Contract No. F49620-82-C-0018. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A150 569.

Key words:

This report describes TWIRL, a simulation of a primarily ground combat engagement between two opposing military forces. It was developed to further experiment with the ROSS language, an object-oriented simulation language that was successfully used to develop the SWIRL air battle simulation, and to develop a prototype simulation that could be used to explore issues in electronic combat. The authors describe the objects that comprise TWIRL and provide extensive examples of object behaviors to explain and illustrate the process of building a

simulation in Ross. (author)

8787 D'Ippolito, Richard; Lee, Kenneth J.; Plinta, Charles; Rissman, Michael S.; Van Scoy, Roger; *AN OOD PARADIGM FOR FLIGHT SIMULATORS, 2ND EDITION.* Report No. CMU/SEI-88-TR-30. 127 p. Sep 1988. Sponsored by SEI-Joint Program Office, Hanscom Air Force Base, Hanscom, MA 01731. Grant/Contract No. F1962885C0003. Avail. from Carnegie Mellon University, Pittsburgh, PA 15213-3890.

Key words:

This report presents a paradigm for object-oriented implementations of flight simulators. It is a result of work on the Ada Simulator Validation Program (ASVP) carried out by members of the technical staff at the Software Engineering Institute (SEI). (author)

8796 Buchanan, Bruce G.; Schoen, Eric; Smith, Reid G.; "DESIGN OF KNOWLEDGE-BASED SYSTEMS WITH A KNOWLEDGE-BASED ASSISTANT," In *IEEE Transactions on Software Engineering.* 14(12): Dec 1988. Report No. IEEE Log Number 8824633. pp. 1771-1791.

Key words:

Intelligent assistants facilitate design and construction of complex software. In this article, the authors propose a model for an intelligent assistant to aid in building one kind of software, knowledge-based systems (KBS), and discuss a preliminary implementation. The assistant participates in KBS construction, including acquisition of an initial model of a problem domain, acquisition of control, and task-specific inference knowledge. The authors present a hypothetical scenario in which the assistant and a KBS designer cooperate to create an initial domain model, and discuss five categories of knowledge the assistant requires to offer such help. The authors then discuss two software technologies upon which the assistant is based: an object-oriented programming language and a user-interface framework. (author)

8914 Gardner, Michael R.; "SUCCESSES AND LIMITATIONS OF OBJECT-ORIENTED DESIGN," In *Journal of Pascal, Ada & Modula 2.* 7(6): Nov 1988. pp. 30-41.

Key words:

This article has two main purposes: (1) to show how to use object-oriented design on a software system sufficiently large that the method must be used recursively through several levels of recursion, and (2) to evaluate the suitability of object-oriented design as a general methodology for decomposing a system into modules. The article's principle example of object-oriented design concerns a hierarchical database management system (DBMS). Accordingly, a secondary purpose of this article will be to discuss some techniques for using Ada to implement a DBMS. (author)

8915 Amir, Shawn; "BUILDING INTEGRATED EXPERT SYSTEMS," In *AI Expert.* 4(1): Jan 1989. pp. 26-37.

Key words:

This article discusses the fundamentals of object-oriented programming in artificial intelligence (AI), especially expert systems. The architecture of Property-list OBjects (POB) and the Common LISP implementation of POB will also be reviewed. Technical material will be presented in sufficient detail to allow implementation and experimentation with POB and Object-Oriented Inference Engine (OBIE) variations. (author)

8916 Jacobs, Jeff; Morgan, Tom; Rettig, Marc; Wimberly, Doug; "OBJECT-ORIENTED PROGRAMMING IN AI - NEW CHOICES," In *AI Expert.* 4(1): Jan 1989. pp. 53-69.

Key words:

This paper describes various software products dealing with object-oriented programming. The authors divided this field into language families: Smalltalks, C derivatives, object-oriented LISPs, and a few languages that do not quite fit into any family, such as Whitewater's Actor. Some guidelines for choosing the best object-oriented language for your particular needs are also

discussed. (author)

8936 Muller, Robert J.; Pircher, Peter A.; Wasserman, Anthony I.; "AN OBJECT-ORIENTED STRUCTURED DESIGN METHOD FOR CODE GENERATION," In *Software Engineering Notes (ACM SIGSOFT)*. 14(1): Jan 1989. pp. 32-55.

Key words:

The overall architecture of a software system has long been recognized as an important contributor to its quality (or lack thereof). Several methods are described that offer valuable concepts to address an architectural design method. But no method makes an adequate distinction between the definition and use of objects, which is essential if one is to develop a library of reusable objects. In addition, the object-oriented methods have largely abandoned Structured Design, which is well established and includes most of the necessary concepts and notation. As a result, the authors decided to synthesize ideas from these methods, along with their own ideas, to define a new method, called Object-Oriented Structured Design (OOSD), for architectural design of systems. (author)

8949 Corradi, Antonio; Leonardi, Letizia; "PO: AN OBJECT MODEL TO EXPRESS PARALLELISM," In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*. Sep 1988. pp. 152-155.

Key words:

Concurrency is actually one of the neglected issues of object systems. The majority of issues simply address processes as instances of a system class. This dichotomy, passive objects/active processes, contrasts with object uniformity. Parallel Objects (PO) proposal is an example of insertion of parallelism in an object framework that follows the principle of uniformity. The PO model is presented in this article as an object model to express parallelism. (author)

8962 Seidewitz, Ed; "GENERAL OBJECT-ORIENTED SOFTWARE DEVELOPMENT: BACKGROUND AND EXPERIENCE," In *Journal of Systems and Software*. 9(2): Feb 1989. pp. 95-108.

Key words:

The effective use of Ada requires the adoption of modern software-engineering techniques such as object-oriented methodologies. A Goddard Space Flight Center Software Engineering Laboratory Ada pilot project has provided an opportunity for studying object-oriented design in Ada. The project involves the development of a simulation system in Ada in parallel with a similar Fortran development. As part of the project, the Ada development team trained and evaluated object-oriented and process-oriented design methodologies for Ada. Finding these methodologies limited in various ways, the team created a general object-oriented development methodology that they applied to the project. This paper discusses some background on the development of the methodology, describes the main principles of the approach, and presents some experiences using the methodology, including a general comparison of the Ada and Fortran simulator designs. (author)

9058 Barry, Brian M.; *OBJECT-ORIENTED SIMULATION OF EW SYSTEMS*. Report No. Technical Note 87-31. 65 p. Dec 1987. Avail. from Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145. Order No. AD-A193 782.

Key words:

Simulations of complex EW systems are difficult to build and virtually impossible to thoroughly validate. As a consequence, most EW systems engineers tend to regard results derived from simulations as suspect, preferring to reply instead on laboratory testing and field trials for performance evaluations. The author suggests that the real problem may be that traditional simulations do not provide the kind of modeling and analysis tools which the systems engineer really needs. In this paper, a prototype for a new kind of EW simulation environment which supports an object-oriented approach to modeling and simulation is described. The author will provide some background information on object-oriented programming, describe the software

architecture of the simulation environment and discuss several examples which illustrate its use.

9132 Bailin, Sidney C.; "AN OBJECT-ORIENTED REQUIREMENTS SPECIFICATION METHOD." In *Communications of the ACM.* 32(5): May 1989. pp. 608-623.

Key words:

This article describes a method of analyzing requirements for object-oriented software. The method is intended to flow smoothly into design by object diagrams, and from there into programming with Ada or another high-level language. The method is intended to serve as an alternative to structured analysis when the use of object-oriented design is foreseen. The authors assume that the analyst who is using this method had a textual statement of requirements for a system available.

9173 Whiting, Mark A.; "CONCEPTUAL OBJECT-ORIENTED DESIGN," In *8th Annual Pacific Northwest Software Quality Conference.* Oct 1990. pp 62-72. Sponsored by Department of Energy. Grant/Contract No. DE-AC06-76RLO-1830. Avail. from PNSQC, P.O. Box 970, Beavertown, OR 97075.

Key words:

Conceptual object-oriented design (COOD) is a methodology that is being used at he Pacific Northwest Laboratory (PNL) to study, plan, specify and document high-level solutions to large-scale information processing problems. COOD embodies aspects of object-oriented program design philosophy (which is being applied to the implementation design of software) to provide enhanced tools and techniques for conceptual design. COOD is targeted at eh phase of software development following requirements analysis and prior to implementation or detailed design. This step is necessary, particularly for large-scale information processing systems to achieve the following: 1. allow designers to conceptually work out solutions to information processing problems where innovative thinking is required; 2. allow a structured environment in which to capture design products, and ; 3. provide a global view of the conceptual solution in an understandable form to the implementors of the solution. This will facilitate their detailed design efforts. The product of COOD is a "Conceptual design specification." This specification is delivered to an implementation team to assist the detailed design process, yet is not a software specification in and of itself. (authors)

9267 Buser, Jon F.; Ward, Paul T.; "REPRESENTING OBJECT ORIENTED SPECIFICATIONS AND DESIGNS WITH EXTENDED DATA FLOW NOTATIONS," In *Proceedings of the 13th Annual Software Engineering Workshop.* Nov 1988.

10020 Foy, Ralph A.; Loftus, William P.; Oei, Charles L.; Thalhamer, John A.; "ADA ABSTRACT DATA TYPES--THE FOUNDATION OF AN INTERACTIVE ADA COMMAND ENVIRONMENT," In *Proceedings of the 7th Annual National Conference on Ada Technology, March 13-16, 1989.* Mar 1989. pp. 326-331.

Key words:

The Ada Command Environment (ACE) is an interactive, object-oriented software development environment. The ACE uses Ada as both the command language and the programming language; and supports an abstract data type (ADT) view of the underlying operating system and applications tools. The benefits of the ACE approach is the combination of ADTs and the Ada programming language. The Ada language provides a strong foundation for the construction and use of ADTs and ADTs provide the mechanism for environment manipulation. (author)

10042 Barlev, S.; Davanzo, P.; Hetzron, J.; Levitz, M.; Tupper, K.; "ADA DESIGN TOOL," In *Proceedings of the 7th Annual National Conference on Ada Technology, March 13-16, 1989.* Mar 1989. pp. 557-566.

Key words:

The Ada Design Tool (ADT) is being designed and integrated into a software engineering environment to support in the conceptualization, preparation and generation of Ada programs. The graphical and textual editors of the ADT allow the software engineer to represent a top-level and detailed-level design in an Object-Oriented Design approach or a Functional Decomposition Design approach. The ADT has a validation function to ensure that the design is complete and consistent, a source code generator which is a facility for generating Ada source code, and a documentation function which produces MIL-STD specifications as well as analytical reports.

10046 Brown, Russell; Dobbs, Verlynda; "A METHOD OF TRANSLATING FUNCTIONAL REQUIREMENTS FOR OBJECT-ORIENTED DESIGN," In *Proceedings of the 7th Annual National Conference on Ada Technology, March 13-16, 1989.* Mar 1989. pp. 589-599.

Key words:

A challenge in the use of Object-Oriented Design methods for software design is the difficulty of maintaining traceability between functional requirements and the object requirements. A framework for translating functional specifications into a set of object requirements, called Functional Requirements Translation (FRT), is presented in this paper. FRT is intended for use of OOD methods for DoD systems developed in Ada. This forms-based methodology provides bi-directional traceability of the translation and "can be used to identify unsatisfied requirements and produce good detailed object designs". (author)

10057 Perez, Eduardo Perez; "SIMULATING INHERITANCE WITH ADA," In *ACM Ada Letters.* 8(5): Sep 1988. pp. 37-46.

Key words:

Since the evolution of object-oriented programming languages and systems, interest in inheritance has increased. Inheritance is a mechanism to help a software designer in the specification of software components. The designer need only indicate that a component inherits the specification of another and specify any differential features between the two. This introduces a new method of software development called differential development of software, or more precisely, incremental development of software. The inheritance mechanism of Smalltalk 80 is reviewed and the different steps taken in the inheritance process. The Ada concept of derived types is analyzed because it facilitates the simulation of an inheritance mechanism similar to the Smalltalk 80 model.

10103 Forestier, J. P.; Fornarino, C.; Franchi-Zannettacci, P.; "ADA++: A CLASS AND INHERITANCE EXTENSION FOR ADA," Jun 1989. pp. 3-15.

Key words:

ADA++ is a superset of Ada supporting the use of object-oriented design and constructs above standard Ada. A full model for class definition and multiple inheritance on abstract objects fully compatible with standard Ada syntax, semantics and methodology is provided. Currently, Ada++ is implemented as a pre-processor and embedded in a graphical interactive programming environment called, ADALOOK.

10104 Donaldson, C. M.; "DYNAMIC BINDING AND INHERITANCE IN AN OBJECT-ORIENTED ADA DESIGN," Jun 1989. pp. 16-25.

Key words:

Classic-Ada is an object-oriented design language and toolset developed under a research and development effort at Software Productivity Solutions, Inc. The language incorporates the standard Ada syntax, semantics and methodology and includes a set of extensions to support dynamic binding and inheritance. A User Interface Management System (UIMS), which is a collection of reusable components for building applications user interfaces, has been designed

using Classic-Ada.

10106 Atkinson, Colin; Bayan, Rami; Cardigno, Cinzia; Destombes, Catherine; Di Maio, Andrea; "DRAGOON: AN ADA-BASED OBJECT ORIENTED LANGUAGE FOR CONCURRENT, REAL-TIME, DISTRIBUTED SYSTEMS," Jun 1989. pp. 39-48.

Key words:

DRAGOON (Distributed Reusable Ada Generated from an Object Oriented Notation) is a fully object oriented design and programming language which can be automatically mapped into Ada for execution. DRAGOON provides inheritance and polymorphism and thus enriches Ada with the typical features of an object-oriented paradigm. It is described in this paper how DRAGOON can be used to design concurrent, distributable and dynamically reconfigurable applications. (author)

10108 Davis, Neil W.; Irving, Malcolm; Lee, John E.; "PRACTICAL EXPERIENCES OF ADA AND OBJECT ORIENTED DESIGN IN REAL TIME DISTRIBUTED SYSTEMS," Jun 1989. pp. 59-79.

Key words:

Logica Space and Defence Systems Limited is currently working to produce an object oriented approach to software system development. Practical experiences in the use of Ada and object oriented design in the requirements analysis and design of real time distributed systems are presented in this paper. Reported are lessons learned and an overview of future work needed in this area.

10110 Auxiette, G.; Cabadi, J. F.; Rehbinder, P.; "PROMETHEE: DESIGNING A PROCESS CONTROL SYSTEM," Jun 1989. pp. 90-104.

Key words:

The "state of the art" of several activities related to Ada, such as, design for Ada, integration in Unix, compatibility with software libraries or networks, are examined in this paper. The authors present solutions to some of these activities, and others are left unsolved. The study is a process control system, called Promethee, and an overview of it is presented. Issues of the design process, especially those related to soft real-time systems are also discussed.

10205 Liu, Chang-Shyan; Yau, Stephen S.; "A STRUCTURED BIPARTITE INHERITANCE NETWORK REPRESENTATION FOR OBJECT ORIENTED SOFTWARE DESIGN," Sep 1989. pp. 351-357. Order No. 0730-3157/89/0000/0351$01.00.

Key words:

In this paper, a representation for any object-oriented software design is presented. The representation is based on a Structured Bipartite Inheritance Network, which is a network with two kinds of basic nodes: data entity nodes and action nodes, and an encapsulation mechanism: substructure. Data entity nodes and action nodes are independent of each other and structured into inheritance hierarchy. The advantage of this representation is that all object-oriented software design can be represented in a uniform way and thus makes the software system more understandable and more maintainable. (author)

10317 Kuhl, Frederick S.; "OBJECT-ORIENTED PROGRAMMING APPLIED TO A PROTOTYPE WORKSTATION," In *Software - Practice and Experience*. 20(9): Sep 1990. Report No. 0038-0644/90/090887-12$0. pp. 887-898.

Key words:

Object-oriented programming has been applied to the development of a prototype workstation to be used in airport traffic control towers. Objective-C was used because it supports objects, classes and inheritance, and it allows easy access to system services. A number of design practices emerged as helpful in the course of development, some of which have been reported elsewhere. The notion of a framework of co-operating classes as a paradigm of design was

especially helpful. Comparisons with the size and rate of code production of an earlier, similar workstation programmed in C indicate an advantage to object-oriented programming. (author)

10400 Grubbs, Jeffrey W.; Roggio, Robert F.; "REUSE BY DESIGN: DATA ABSTRACTION VS. THE 'TOP-DOWN' MINDSET IN AN OBJECT-ORIENTED ENVIRONMENT," In *5th Annual Knowledge-Based Software Assistant Conference*. Sep 1990. pp.418-431.

Key words:

Utilization of an object-oriented environment for application development is no guarantee that software produced therein will be designed for reusability. Designer/programmer methodological bias coupled with a misunderstanding of the object-oriented approach produces marginally effective abstractions with ill-defined or inadequate behaviors. These inappropriate abstractions tie software components to a specific application context and severely restrict the opportunities for reuse. This paper examines a limited Smalltalk application whose development demonstrates such effects. The qualitative and quantitative advantages realized through redesigning it for reusability are also discussed. (author)

10414 Hoffman, Daniel; "ON CRITERIA FOR MODULE INTERFACES," In *IEEE Transactions on Software Engineering*. 16(5): May 1990. pp 537 - 542. Grant/Contract No. A8067. Sponsored by Natural Sciences and Engineering Research Council of Canada. Order No. 0098-5589/90/0500-0537$01.00.

Key words:

While the benefits of modular software development are widely acknowledged, there is little agreement as to what constitutes a good module interface. Computational complexity techniques allow us to evaluate algorithm time and space costs but offer no guidance in the design of the interface to an implementation. Yet, interface design decisions often have a critical effect on the development and maintenance costs of large software systems. In this paper the author presents criteria that have led to simple, elegant interfaces. These criteria have been developed and refined through repeated practical application. The author presents and illustrates the criteria in detail.

10624 Meyer, Bertrand; "LESSONS FROM THE DESIGN OF THE EIFFEL LIBRARIES," In *Communications of the ACM*. 33(9): Sep 1990. pp 69-88. Order No. ACM 001-0782/90/0900-0069$1.50.

Key words:

The use of reusable software components is now technically possible and should advance the level of software development. This article presents the efforts which have been made to advance the cause of component based software development in the Eiffel environment through the construction of the Basic Eiffel Libraries. Following a brief overview of the libraries, this article reviews the major language techniques that have made them possible (with more background about Eiffel). It then discusses design issues for libraries of reusable components, the use of inheritance hierarchies, the indexing problem, and planned developments. (author)

10636 Wirfs-Brock, Rebecca J.; Johnson, Ralph E.; "SURVEYING CURRENT RESEARCH IN OBJECT-ORIENTED DESIGN," In *Communications of the ACM*. 33(9): Sep 1990. pp 104-124. Order No. ACM 001-0782/90/0900-0104$1.50.

Key words:

The state of object-oriented design is evolving rapidly. This survey describes what are currently thought to be the key ideas, necessarily incomplete, of both academic and industrial efforts in both the United States and Europe. It ignores well known ideas like those of Coad and Meyer in favor of less widely known projects. Presented are separate works by Alan Snyder and Dennis de Champeaux of Hewlett-Packard, Rebecca Wirfs-Brock from Tektronix, Ralph Johnson at the University of Illinois, and results from the research group in object oriented software engineering led by Karl Lieberherr at Northeastern University. It is found that standardization of terminology is needed, however the fact that different groups are forced to invent terminology

for the same concepts are important. The various methods presented tended to complement each other rather than compete, with their similarities hidden in differences in vocabulary.

10668 Dony, Christophe: "EXCEPTION HANDLING AND OBJECT-ORIENTED PROGRAMMING: TOWARDS A SYNTHESIS," In *ECOOP/OOPSLA 1990 Proceedings*. 25(10): Oct 1990. pp 322-330. Sponsored by Rank-Xerox & LITP. Order No. ACM089791-411-2/90/0010-0322$1.50.

Key words:

The paper presents a discussion and a specification of an exception handling system dedicated to object-oriented programming. The authors show how a full object-oriented representation of exceptions and of protocols to handle them, using meta-classes, makes the system powerful as well as extendible and solves many classical exception handling issues. The authors explain the interest for object-oriented programming of handlers attached to classes and to expressions. They propose an original algorithm for propagating exceptions along the invocation chain which takes into account, at each stack level, both kind of handlers. Any class can control which exceptions will be propagated out of its methods; any method can provide context-dependent answers to exceptional events. The whole specification and some keys of the author's Smalltalk implementation are presented in the paper. (authors)

10799 Moreau, Dennis R.; Dominick, Wayne D.; "A PROGRAMMING ENVIRONMENT EVALUATION METHODOLOGY FOR OBJECT-ORIENTED SYSTEMS: PART I- THE METHODOLOGY," In *Journal of Object-Oriented Programming*. 3(1): May 1990. Report No. ISSN #0896-8438. pp. 38-54.

Key words:

The research presented in this article addresses the design, development, and evaluation of a systematic, extensible, and environment-independent methodology for the comparative evaluation of object-oriented programming environments. This methodology is the intended to serve as a foundation element for supporting research into the impact of object-oriented software development environments and design strategies on the software development process and resultant software products.(author)

10817 Cox, Brad J.; *OBJECT ORIENTED PROGRAMMING: AN EVOLUTIONARY APPROACH.* Report No. ISBN 0-201-10393-1. 285 p. Avail. from Addison-Wesley Publishing Company. Order No. ISBN 0-201-10393-1.

Key words:

This book describes Object-Oriented Programming (OOP). The focus is on OOP not so much as a coding technique, but as a code packaging technique, a way for code suppliers to encapsulate functionality for delivery to consumers. Inheritance and encapsulation are the major existing modules. The initial chapters describe the system-building problem. The middle chapters describe a solution as implemented in Objective-C. Although encapsulation and inheritance provide the technical underpinnings for large-scale reusability, they are useless without an information network and libraries for reusability. The closing chapter describes some techniques for extending the basic object-oriented definitions to handle even more ambitious problems, including automated garbage collection, heap compaction, virtual object memories, and distributed systems.

10818 Jackson, Michael; *SYSTEM DEVELOPMENT.* Report No. ISBN 0-13-880328-5. 435 p. Avail. from Prentice-Hall, Rt. 59 at Brook Hill Drive, West Nyack, NY 10995. Order No. ISBN 0-13-880328-5.

Key words:

This book is about Jackson System Development (JSD), a system development method especially oriented towards systems in which time is important. Part I is an overview of JSD. Part II gives a detailed description of each major step in JSD in a separate chapter. These steps are illustrated with three example problems. JSD consists of six steps, the first four concerned with specification and the last two with implementation. What is often called design

has largely been absorbed into the implementation steps. JSD begins by constructing a model of the real world concentrating on the entities with which the system will be concerned, their actions, and their orderings in time. A JSD user worries about function only after this model has been defined. Part III considers various topics, namely the input system and errors, system maintenance, and a retrospective look at JSD.

10819 Booch, Grady; *SOFTWARE COMPONENTS WITH ADA: STRUCTURES, TOOLS, AND SUBSYSTEMS.* Report No. ISBN 0-8053-0610-2. 665 p. Avail. from Benjamin Cummings. Order No. ISBN 0-8053-0610-2.

Key words:

A carefully engineered collection of reusable components can reduce the cost of software development, improve the quality of software products, and accelerate software production. This book is designed to train the reader in the creation and application of such components. It provides a catalog of reusable software components, illustrates how each component was developed, and demonstrates how they collectively can be applied to the construction of complex systems. The components are implemented in Ada and illustrate the use of object-oriented techniques. A scheme for classifying components is presented.

10820 Meyer, Bertrand; *OBJECT-ORIENTED SOFTWARE CONSTRUCTION.* Report No. ISBN 0-13-629049-3. 552 p. Avail. from Prentice-Hall, Rt. 59 at Brook Hill Drive, West Nyack, NY 10995. Order No. ISBN 0-13-629049-3.

Key words:

This book describes Object Oriented Design. Part 1 presents the problems OOD is meant to solve and gives a high-level argument of why one should structure systems around data instead of functions. OOD's support for certain principles of modularity is discussed. OOD is defined as the construction of software systems as structured collections of abstract data type implementations. Part 2 consists of a detailed explanation of OOD. The programming language Eiffel is used as a notation for conveying Object-Oriented principles. Part 3 discusses how to implement OOD in other languages. Classical languages, namely C, Fortran, and Pascal; Ada; and other Object-Oriented languages, namely Simula, Smalltalk, C++, Objective C, and Lisp variants are all treated. The book concludes with a brief indication of issues for further research. Appendices summarize technical details about Eiffel such as the library, the grammar, reserved words, Input/Output, and syntax diagrams.

10821 Wiener, Richard S.; Pinson, Lewis J.; "A PRACTICAL EXAMPLE OF MULTIPLE INHERITANCE IN C++," In *ACM SIGPLAN Notices.* 24(9): Sep 1989. pp. 112-115.

Key words:

Version 2.0 of C++ supports multiple inheritance, which offers an object oriented designer an additional degree of freedom. If used in a disciplined way, it can simplify an inheritance hierarchy. If abused, it can add tremendous complexity to a software design, perhaps in the extreme making it unmanageable. This article presents as an example an appropriate use of multiple inheritance, an array of integers. Integers, arrays, and integer arrays are all classes in the example.

10822 Meyer, Bertrand; "GENERICITY VERSUS INHERITANCE," In *ACM SIGPLAN Notices.* 21(11): Nov 1986. In *OOPSLA 1986 Conference Proceedings.* Sep 1986. pp. 391-405.

Key words:

Genericity, as in Ada or ML, and inheritance, as in object-oriented languages, are two alternative techniques for ensuring better extendibility, reusability, and compatibility of software components. This article is a comparative analysis of these two methods. It studies their similarities and differences and assesses to what extent each may be simulated in a language offering only the other. It shows what features are needed to successfully combine the two approaches in a statically typed language and presents the main features of the programming language Eiffel, whose design, resulting in part from this study, includes multiple inheritance and

a limited form of genericity under full static typing. (Author)

10823 Stevens, Al; "FROM C TO C++," In *Dr. Dobb's Journal.* Dec 1989. pp. 8-17.

Key words:

This article consists of two interviews, one with Dennis Ritchie, the designer of C, and the other with Bjarne Stroustrup, the creator of C++. C++ is an object oriented superset of the programming language C. Both interviews present their subjects views on the history, current activities, and future prospects of their respective languages. Ritchie is asked for his opinion on the American National Standards Institute's standard for C. Stroustrup is asked about recent and future Personal Computer implementations of C++, as well as the development of his recent version 2.0.

10824 Seidewitz, Ed; Stark, Michael; "ADA IN THE SEL: EXPERIENCES WITH OPERATIONAL ADA PROJECTS," In *Proceedings of the Second NASA Ada User's Symposium.* Nov 1989. 11 p.

Key words:

This set of slides surveys Ada projects conducted in the flight dynamics division at NASA's Goddard Space Flight Center (GSFC). Eight projects have been performed, and data has been collected by the Software Engineering Laboratory for all of them. As programmers have gained more experience with Ada, they tend to design a greater proportion of generic packages, more types, and less tasks. Packages tend to become smaller. Ada projects tended to reuse more code than Fortran projects, the traditional language at NASA/GSFC. Several Ada projects had to be performed before design errors decreased to the same order as with Fortran projects. Similar results hold for errors due to previous changes. On the other hand, interface errors for Ada projects were always less than for Fortran projects, and continually declined. This talk presents data supporting these and additional conclusions.

10825 Stein, Lynn Andrea; "DELEGATION IS INHERITANCE," In *OOPSLA 1987 Proceedings.* Oct 1987. In *ACM SIGPLAN Notices.* 22(12): Dec 1987. pp. 138-146.

Key words:

Inheritance and delgation are alternate methods for incremental definition and sharing. It has commonly been believed that delegation provides a more powerful model. This paper demonstrates that there is a natural model of inheritance which captures all of the properties of delegation. Independently, certain constraints on the ability of delegation to capture inheritance are demonstrated. Finally, a new framework which fully captures both delegation and inheritance is outlined, and some of the ramifications of this hybrid model are explored. (Author)

10826 Cointe, Pierre; "METACLASSES ARE FIRST CLASS: THE OBJVISP MODEL," In *OOPSLA 1987 Proceedings.* Oct 1987. In *ACM SIGPLAN Notices.* 22(12): Dec 1987. pp. 156-167.

Key words:

This paper shows how an attempt at an uniform and reflective definition resulted in an open-ended system supporting ObjVlisp, which is used to simulate object-oriented language extensions. The author proposes to unify Smalltalk classes and their terminal instances. This unification allows one to treat a class as a "first class citizen," to give a circular definition of the first metaclass, to access to the metaclass level, and finally, to control the instantiation link. Because each object is an instance of another one and because a metaclass is a real class inheriting from another one, the metaclass links can be created indefinitely. This uniformity allows one to define the class variables at the metalevel thus suppressing the Smalltalk-80 ambiguity between class variables and instance variables: in this paper's model the instance variables of a class are the class variables of its instance. (Author)

10827 Minsky, Naftaly H.; Rozenshtein, David; "A LAW-BASED APPROACH TO OBJECT-ORIENTED PROGRAMMING," In *OOPSLA 1987 Proceedings.* Oct 1987. In *ACM SIGPLAN Notices.* 22(12): Dec 1987. pp. 482-493.

Key words:

The central idea behind this paper is that the discipline governing the exchange of messages between objects should be specifiable by the programmer in the form of an explicit law of the system. The authors show how, starting from a very primitive foundation, which presumes neither encapsulation nor inheritance, one can establish various forms of both, as well as other useful disciplines, simply by means of appropriate laws. (Author)

10828 Abbott, Russell J.; "PROGRAM DESIGN BY INFORMAL ENGLISH DESCRIPTIONS," In *Communications of the ACM.* 26(11): Nov 1983. pp. 882-894.

Key words:

A technique is presented for developing programs from informal but precise English descriptions. The technique shows how to derive data types from common nouns, variables from direct referencs, operators from verbs and attributes, and control structures from their English equivalents. The primary contribution is the proposed relationships between common nouns and data types; the others follow directly. Ada is used as the target programming language because it has useful program design constructs.

10829 Backus, John; "CAN PROGRAMMING BE LIBERATED FROM THE VON NEUMANN STYLE? A FUNCTIONAL STYLE AND ITS ALGEBRA OF PROGRAMS," In *Communications of the ACM.* 21(8): Aug 1978. pp. 613-641.

Key words:

Conventional programming languages are growing ever more enormous, but not stonger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor - the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs. An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages. Associated with the functional style of programming is an algebra of programs whose variables range over programs and whose operations are combining forms. This algebra can be used to transform programs and to solve equations whose "unknowns" are programs in much the same way one transforms equations in high school algebra. These transformations are given by algebraic laws and are carried out in the same language in which programs are written. Combining forms are chosen not only for their programming power but also for the power of their associated algebraic laws. General theorems of the algebra give the detailed behavior and termination conditions for large classes of programs. A new class of computing systems uses the functional programming style both in its programming language and in its state transition rules. Unlike von Neumann languages, these systems have semantics loosely coupled to states - only one state transition occurs per major computation.

10830 Booch, Grady; "OBJECT-ORIENTED DEVELOPMENT," In *IEEE Transactions on Software Engineering.* 12(2): Feb 1986. pp. 211-221.

Key words:

Object-oriented development is a partial lifecycle software development method in which the decomposition of a system is based upon the concept of an object. This method is fundamentally different from traditional function approaches to design and serves to help manage the complexity of massive software-intensive systems. This paper examines the process of object-oriented development as well as the influences upon this approach from advances in abstraction mechanisms, programming languages, and hardware. The concept of an object is central to object-oriented development, and so the properties of an object are

discussed in detail. The paper concludes with an examination of the mapping of object-oriented techniques to Ada using a design case study. (Author)

10831 Reynolds, Charles W.: "ON IMPLEMENTING GENERIC DATA STRUCTURES IN MODULA-2." In *Journal of Pascal, Ada & Modula 2.* 6(5): Sep 1987. pp. 26-38.

Key words:

A generic data structure is a data type for which the set of operations is specified, but not the set of values. Generics should ideally exhibit strong type checking, information hiding, separate compilation of interfaces and implementations, and efficiencies of both time and space. The programming language Modula-2 provides strong support for data abstraction, but no obvious capability for defining generics. This article briefly reviews past proposals for simulating generics in Modula-2 and proposes a new solution. The new solution uses a new statement, the "Include" statement, which can be added with a preprocessor.

10832 *DOD-STD-2167A MILITARY STANDARD DEFENSE SYSTEM SOFTWARE DEVELOPMENT.* Report No. DoD-STD-2167A. 61 p. Feb 1988. Avail. from Data & Analysis Center for Software, P.O. Box 120, Utica, NY 13503. Order No. DoD-STD-2167A.

Key words:

This Department of Defense standard, along with the accompanying Data Item Descriptions (DIDs), establishes uniform requirements for the acquisition, development, or support of software systems. These requirements apply to the development of Computer Software Configuration Items (CSCIs), including the software element of firmware. The requirements of this standard lie in the areas of software developent management, software engineering, formal qualification testing, software product evaluation, software configuration managment, and transitioning to software support.

## Title Key-Word-in-Context List

## Author Index