View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Calhoun, Institutional Archive of the Naval Postgraduate School

MONTEREY, CALIFORNIA 93940

NPS52-88-044

# NAVAL POSTGRADUATE SCHOOL

## Monterey, California

# OBJECT-ORIENTED RAPID PROTOTYPING

by Valdis Berzins

September 1988

Approved for public release; distribution is unlimited.

Prepared for:

Naval Postgraduate School
Monterey, CA 93943

FEDDOCS
D 208.14/2
NPS-52-88-044

FedDocs
D 208.14/2:NPS-52-88-044

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

Harrison Shull
Provost

The research reported herein was conducted with funds administered by the Naval Postgraduate School Research Council.

Reproduction of all or part of this report is authorized.

This report was prepared by:

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>UNCLASSIFIED | 1b RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY<br>2b. DECLASSIFICATION / DOWNGRADING SCHEDULE | 3 DISTRIBUTION / AVAILABILITY OF REPORT<br>Approved for public release; distribution is unlimited. |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>NPS52-88-044 | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION<br>Naval Postgraduate School<br>6b. OFFICE SYMBOL (If applicable)<br>52 | 7a. NAME OF MONITORING ORGANIZATION<br>Naval Postgraduate School |
| 6c. ADDRESS (City, State, and ZIP Code)<br>Monterey, CA 93943 | 7b. ADDRESS (City, State, and ZIP Code)<br>Monterey, CA 93943 |
| 8a. NAME OF FUNDING / SPONSORING ORGANIZATION<br>Naval Postgraduate School<br>8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER<br>O&MN, Direct funding |
| 8c. ADDRESS (City, State, and ZIP Code)<br>Monterey, CA 93943 | 10. SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO. |

11. TITLE (Include Security Classification)
OBJECT-ORIENTED RAPID PROTOTYPING (U)

12. PERSONAL AUTHOR(S) BERZINS, Valdis

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year, Month, Day) | 15 PAGE COUNT |
|---|---|---|---|
| Progress | FROM 87/10 TO 88/9 | 1988 September | 17 |

16. SUPPLEMENTARY NOTATION

| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | Rapid prototyping, objects, abstractions, reusable components, CASE, engineering database |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)


Object-oriented techniques form a promising approach for realizing an integrated computer-aided prototyping environment capable of detecting and correcting errors early in the development process. We discuss the connection between rapid prototyping, object-oriented data models, formal specifications, reusable components, and engineering databases.


| | |
|---|---|
| 20. DISTRIBUTION / AVAILABILITY OF ABSTRACT<br>☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION<br>UNCLASSIFIED |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL<br>Valdis Berzins | 22b TELEPHONE (Include Area Code) (408)646-2461 / 22c. OFFICE SYMBOL 52Be |

DD FORM 1473, 84 MAR    83 APR edition may be used until exhausted.
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE


☆ U.S. Government Printing Office: 1986—606-24.


UNCLASSIFIED

# Object-Oriented Rapid Prototyping

*Valdis Berzins*

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Berzins@NPS-CS.arpa
(408) 646-2461

*ABSTRACT*

Object-oriented techniques form a promising approach for realizing an integrated computer-aided prototyping environment capable of detecting and correcting errors early in the development process. We discuss the connection between rapid prototyping, object-oriented data models, formal specifications, reusable components, and engineering databases.

**Keywords**: Rapid prototyping, objects, abstractions, reusable components, CASE, engineering database.

## 1. Introduction

The term "object-oriented" has not been precisely defined, and in common usage its meaning varies somewhat with the context. The oldest application of object-oriented techniques has been to programming languages. A survey of the current state of the art in object-oriented programming can be found in [7]. Other applications of object-oriented techniques include specification languages, conceptual modeling, and engineering databases. All object-oriented techniques involve abstract objects whose interactions are limited to explicit interfaces. Such objects provide a means for localizing information, and many applications of object-oriented techniques focus on localizing state information. Object-oriented approaches also commonly organize objects into hierarchies with an inheritance mechanism, although this is not universal. Localized information and inheritance are independent concepts which can be usefully combined in many contexts.

There is a close correspondence between objects and abstract data types. Objects are viewed as individuals that communicate via messages. An abstract data type consists of a set of instances and a set of operations involving those instances. An instance of an abstract data type corresponds to an object and the operations of an abstract data type correspond to the messages recognized by the object.

Abstract data types can be classified as mutable or immutable according to whether or not the the properties of the type are subject to change. The instances of a mutable type can be created, modified, and destroyed, while the instances of an immutable type form a fixed set of values, where each value has fixed properties. Work on programming languages supporting abstract data types, such as CLU and Ada, has treated both mutable and immutable types. While there has been some work on the theory of mutable abstract data types, most of the work on the theory of abstract data types has concentrated on immutable types [8, 9], and narrow interpretations of the term "abstract data type" have sometimes excluded the mutable variety. Work on object-oriented programming has tended to emphasize mutable classes of objects, even though immutable objects are useful in some applications. Thus it appears to be an oversimplification to say that the the instances of an object class always have internal states while the instances of an abstract data type never have internal states.

The distinction between abstract data types and objects is the point of view from which the behavior of a type or a class of objects is described. The behavior of an abstract data types is described in terms of operations that can be applied to the instances of the type, while the behavior of a class of objects is generally described in terms of the methods used by the objects to respond to the messages they receive. This difference is largely cosmetic, but it has lead to emphasis on different aspects of object behavior. The simplest kinds of abstract data types identify operations with functions on the instances. In object-oriented approaches objects are viewed as potentially active agents that can respond to messages, which naturally leads to a models where objects can perform actions independently and concurrently [1, 5]. Object oriented approaches have also raised the question of how to handle classes of objects that all provide the same external behavior but where different instances of the class can have different internal data representations and different algorithms for implementing corresponding methods.

To be effective aids in requirements analysis, prototypes must be constructed and modified rapidly and at low cost. The limiting factor in prototyping is the effort of skilled analysts and designers, who are generally scarce and highly paid. The key elements for supporting rapid prototyping are reusable components, abstractions, orthogonal decompositions, and computer aid. Reusable components support rapid prototyping by avoiding repeated effort. Abstractions reduce the amount of detail the analysts and designers must consider, thereby reducing their work load. Orthogonal decompositions limit interactions

between different parts of a specification or design, thus reducing the number of components that must be considered to understand or modify an aspect of the proposed system. Computer-aided techniques can be used to help analysts and designers by automating the routine parts of their tasks and reducing the amount of detail they must consider.

Developing effective means for rapid prototyping is one of the major challenges faced by current research efforts in software engineering. One of the trends in software engineering has been to shift from traditional control-oriented approaches to system description to object-oriented approaches. Such approaches can contribute to all four of the key elements for supporting rapid prototyping identified above. This paper explores some of the object-oriented techniques applicable to rapid prototyping and evaluates their usefulness.

Object-oriented approaches are useful in prototyping because they can be used to simplify and modularize descriptions of complex systems. Localizing information and limiting interactions between different parts of the system is essential for rapid analysis, synthesis, or modification of a prototype design. Object-oriented techniques are important in software development because they reflect the shift from a single processor environment to a distributed, multi-processor environment. An object-oriented viewpoint exposes the parallelism inherent in a problem, and allows simpler descriptions of many problems by allowing logically separate activities to be defined independently of each other, rather than forcing their combination into a single sequential process. This is especially important in the design of distributed and real-time systems, in which nonlocal interactions can lead to interference, and systems have complex behaviors even with respect to carefully modularized descriptions. One of the reasons implementations of real-time systems are often hard to understand is that operations from logically unrelated tasks must be interleaved in order to meet real-time deadlines.

Most of the effort in the early stages of software development is spent on building models. Objects are used to express these models which consist of collections of related objects. A model of the environment for the proposed system is usually developed in the requirements analysis phase. A model of the external interfaces of the system is constructed in the functional specification phase [2], while models of the internal interfaces are constructed in the architectural design phase. These models are the basis for the software tools in advanced computer-aided software development environments. Such environments

depend on engineering database support for effective tool integration and coordination of the concurrent activities of a design team [10]. Object-oriented methods are important for developing models of software systems and the automated tools supporting the construction and analysis of those models.

### 2. Object-Oriented Domain Models

Prototyping is an aid rather than a replacement for requirements analysis. Even when following a prototyping approach to software development, it is necessary to develop a conceptual model of the problem domain to support communication and to aid in identifying the objectives of the proposed system. The domain model developed during requirements analysis can be conveniently expressed in an object-oriented manner. The domain model supplies the concepts needed for describing the world in which the proposed system will operate. These concepts consist of the types of objects in that world, the attributes of those objects, the relations between those objects, and the laws governing those objects and relations. The domain model is important because it forms the basis for all agreements between the customer and the developer.

A type is a set of objects, called the instances of the type, which are all subject to the same attributes, relationships, and laws. Types include object classes familiar from mathematics, such as numbers and sets, as well as object classes from the application domain, such as flights and airports. An attribute is a single-valued mapping from one or more types to another type. A relationship is a mathematical relation, corresponding to a predicate which is true for exactly those n-tuples contained in the relation. The number of tuples in a relation need not be finite. The description of a model consists of a finite number of explicitly declared types, attributes, relationships, and laws. Laws are relationships that must be true for all possible n-tuples of objects from the given types.

The domain model presents a simplified view of the world containing only a fixed set of types, attributes, and relationships. The analysts arrive at such a model by including only those aspects of the world relevant to the customer's problem and the proposed software system for solving that problem. This includes both the aspects of the world that will have to be represented in the proposed software, and the aspects that impose external constraints on the system and motivate the customer's requirements. The domain model includes the known laws governing the behavior of the environment, which can be either

descriptive or prescriptive. Laws can be inherited from general purpose types and relationships in the model library by means of a subclass mechanism.

The objects in a domain model are partially specified. A domain model specifies the properties of the objects in the domain and the constraints they must satisfy, but does not prescribe the dynamic behavior of the objects. Domain models are used in the preliminary stages of requirements analysis for the following purposes.

(1) Communication: the domain model is used as a basis for describing properties and objectives of the proposed system.

(2) Analysis: the domain model is used as a basis for answering questions about the properties of the domain.

(3) Design: the domain model is used as a basis for generating the skeleton of a prototype design for the proposed system.

An example of a fragment of a domain model for an elevator system is shown in Fig. 1. This model introduces some of the types of objects appearing in the elevator domain, together with some of the relationships important for describing the domain. The relationships express properties relating groups of objects. Relationships often have informal interpretations, which are used for relating the formal model to aspects of the real world. Such informal descriptions are important because one of the major activities in requirements analysis is bridging the gap between informal descriptions and formal ones. Note that relationships

---

```
type floor

type elevator
 relation stopped_at(elevator, floor)
   -- true if the elevator is currently stooped at the floor.
 relation doors_open(elevator, floor)
   -- true if the outer doors of the elevator on the floor are currently open.
 law for all(e: elevator, f: floor :: doors_open(e, f) => stopped_at(e, f) )
   -- the doors can only be open if the elevator is stopped on the floor.
```

**Fig. 1 Domain Model Components for an Elevator System**

---

are used as logical predicates rather than as tables for storing facts in a database. The law expresses a constraint (or *invariant*) that corresponds to a specific requirement for passenger safety. Laws can also be used to give formal descriptions of the semantics of relationships by describing connections between different relationships, and can be used to record dependencies between primitive concepts and derived concepts.

Attributes and relationships associated with an object represent its observable properties, and correspond to messages recognized by the object. Since relationships are not necessarily finite, logic programming rather than relational database technology is the natural context for constructing tools for analyzing domain models. The purpose of such tools is to answer questions about the problem domain rather than to simulate the behavior of the proposed system. The objects in the domain model are incomplete because they have methods for observing but not for modifying the states and properties of the objects. The objects in the domain model can form the basis for an executable prototype if they are augmented with methods for initializing and updating the states of the objects to reflect expected interactions between the proposed system and its environment.

### 3. Object-Oriented Specifications

One of the primary difficulties in the development of large software systems is conceptual complexity. Conceptual complexity can be reduced by constructing a set of independent abstractions to describe a complex concept or system. Abstractions are concepts that can be treated as black boxes [3]. A concept qualifies as an abstraction if it can be understood, specified, and analyzed independently of the mechanism used in its implementation. Formal specifications are essential for the effective use of abstractions, since the specifications must be precise to allow use of the abstraction without the need to examine its implementation. Formally defined notations for specifying black boxes are essential for achieving a high degree of automation [6]. The promise of reducing the incidence of errors in the critical early stages of software development is the driving force behind the trend towards automation and formal methods. Errors in requirements analysis, functional specification, and architectural design become progressively more expensive to correct in the later stages. Object-oriented approaches can make the formal methods easier to use and automate.

### 3.1. Object-Oriented Specification Languages

Specification and prototyping languages are important for computer-aided software engineering [6]. Specification languages are designed for the simple description of complex behavior, with automated tools for synthesis and error checking emphasized over the ability to execute the entire language. Prototyping languages are designed to be executable, with simplicity of expression emphasized over execution efficiency. Both kinds of languages can benefit from an object-oriented approach. Some examples of object-oriented specification languages are Spec [5] and MSG [2]. An example of an object-oriented prototyping language is PSDL [11].

### 3.2. Objects in Formal Specifications

A natural and convenient approach to formal specifications is based on an object-oriented event model of computation [5]. In the event model, computations are described in terms of modules, events, and messages. A module is a black box that interacts with other modules only by sending and receiving messages. An event occurs when a message is received by a module at a particular instant of time. A message is a data packet that is sent from one module to another. Modules and messages are the most important kinds of objects in the event model.

Modules can be used to model external systems such as users and peripheral hardware devices, as well as software components. Modules are active black boxes, which have no visible internal structure. The behavior of a module is specified by describing its interface. The interface of a module consists of the set of stimuli it recognizes and the associated responses. A stimulus is an event, and the response is the set of events directly triggered by the stimulus. The events in the response consist of the arrivals of the messages sent out by the module because of the stimulus.

An example of an event in a cruise control system for a car is shown in Fig. 2. This event corresponds to the arrival of a "brake_pressed" message at a module representing the cruise control system.

Messages can be used to model user commands and system responses. Messages represent abstract interactions that can be realized in a wide variety of ways, including procedure call, return from a procedure, Ada rendezvous, coroutine invocation, external I/O, assignments to non-local variables, hardware interrupts, and exceptions.

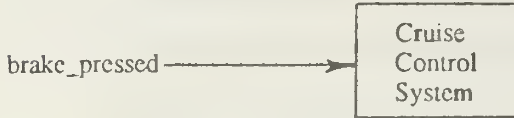



Fig. 2 An Event in a Cruise Control System

Another kind of object useful in formal specification is the *concept*. A concept is a predicate, function, constant, or type that is needed to *describe* the behavior of a module. Concepts are one level removed from the objects in the model, because they appear only in explanations of the required behavior, rather than acting as direct participants in interactions with the proposed system or module. Modules and messages are objects contained in the model of the system, which concepts are meta-objects for that model. Concepts are important for organizing complex descriptions into independent pieces small enough to be readily understood. Concepts are part of the specification language rather than the underlying event model. Concepts are immutable, since their meaning remains the same in all possible states of the proposed system.

### 3.3. Object Hierarchies

Hierarchical structures are important for making complex systems manageable. A popular approach to functional specification uses data flow diagrams to decompose a system into sub-processes, until a level is reached where the processes are easy to describe in terms of a fixed set of primitives. The behavior of a process is described directly only for processes that are not decomposed further. A more recent approach uses abstraction rather than decomposition to arrive at simple descriptions of a proposed system [3, 5]. The advantage of such an approach is a reduction in the number of components in the specification that must be examined to understand any particular interaction between the proposed system and its environment, especially for large systems. This reduction is accomplished by using higher level primitives and user defined abstractions to provide direct descriptions of more complex modules, and by suppressing internal details of the computation at the functional specification stage. Consideration of such details is delayed until the architectural design stage, which is concerned with decomposing computations into good implementation structures.

The abstraction approach does not treat large functional specifications as monolithic entities, however. Black-box specifications are divided into simpler pieces in several qualitatively different ways.

(1) Very large systems often contain a number of nearly independent major subsystems, which have minimal interactions with each other. For example, a spacecraft may have a navigation subsystem, a communication subsystem, and a subsystem for controlling an exploration robot. Such major subsystems should be modeled as distinct central modules, especially if they are going to be assigned to different subcontractors.

(2) Each major subsystem typically has more than one interface. For example, the navigation subsystem may have interfaces with the pilot and with a number of different sensors. Each interface of a central module is specified as a separate view.

(3) The description of each interface of a module is partitioned based on the messages it accepts, the normal and exceptional responses to each message.

(4) The concepts needed to describe the behavior of each message are described by a structured set of definitions. In complex systems the definitions of these concepts can have a hierarchical structure, in which the more abstract concepts are defined in terms of more primitive ones at several levels of detail. The concept hierarchy replaces the data dictionaries used in earlier approaches, and is more general because it includes predicates and functions in addition to data types and constants.

(5) The individual events of a system can be organized into atomic transactions to describe the degree of interleaving allowed between concurrent interactions. The transactions in a complex protocol can also be defined hierarchically.

Black-box specifications of large systems are partitioned by subsystems, interfaces, messages, and responses to show the structure of the system's functionality. The concept hierarchy and the transaction hierarchy impose structure on other aspects of the specification.

Object-oriented models usually include subclass hierarchies with inheritance. Inheritance is an important feature for specification languages which has several important uses in specifying large software systems.

(1) Inheritance can be used to standardize command interfaces across the subsystems of a large system. Consistency in the interpretations of similar commands in different subsystems is an important factor in making large systems easier to use. Specifying subsystem interfaces as subclasses of an object type defining the standard system-wide interface conventions provides a way to specify and mechanically check conformance to such conventions in large systems.

(2) Inheritance provides a mechanism for specializing reusable fragments of specifications. A library of reusable specification components should contain partially specified general purpose building blocks that can be tailored to the needs of each application.

(3) Multiple inheritance can be used to support view integration for systems with multiple interfaces. If each interface of the system as a separate module legibility is improved and concurrent development of different interfaces by different people is enhanced. The entire system is specified by a module that simultaneously inherits the details of the individual interface specifications.

(4) Inheritance can be used to record the steps in a stepwise refinement, where refinements correspond to subclass elaborations. This is especially useful for maintaining the distinction between the details in the user's views of the system and the details in the implementor's view that should not be visible to the users.

### 3.4. Reusable components

Reusable components are useful only if they are applicable in many different situations. This means that they must have coherent and commonly useful functions, and that they must be loosely coupled to their environment, so that they can be readily adapted to many different contexts. Objects are natural candidates for reusable components, since they are abstract and have interactions only through clearly defined interfaces.

Generalization is important for increasing the chances of reusing a particular object. This suggests making objects generic and providing parameters for tailoring their behavior, so that a single component in a software base can be used with many variations, corresponding to different actual parameters.

Inheritance is another important means for enhancing reusability of objects. Fragments of object behavior can be defined as separate views of an objects, which can be combined in different ways via

inheritance. This is another way of providing a greater variety of behaviors from a smaller set of basis elements. Traditionally inheritance has been used at a large scale, to combine the sets of methods supported by different classes of objects. In the context of tailoring the behavior of reusable components, it is also useful to combine features of several different versions of the same method. In the context of specifications, it is easy to use inheritance to combine partial constraints on the behavior of a method from several different ancestors. This is more difficult for programs, but some progress in this direction has been made [4].

### 4. Object-Oriented Databases

Specialized engineering databases are essential for integrated, flexible CAD systems in general [10] and for computer-aided software engineering in particular. Such databases are used for maintaining the versions of the software being developed as well as the library of reusable components and the knowledge bases used by expert systems embedded in the CAD tools. The required capabilities can best be provided by object-oriented databases, which were developed primarily to support engineering applications. Since knowledge of available components is necessary for bottom-up design, this implies a system for managing a very large software base must be responsible for performing bottom-up design functions, rather than the designer [12].

The essential problem in the organization of object-oriented databases for managing reusable components is to allow the representation and retrieval of an unbounded number of components with finite memory and processor speed. An unbounded number of components must be considered because software designs can contain arbitrary user-defined abstract data types, and the reusable components in the component database must be applicable to all of the types in this infinite set to be useful. It is also necessary to allow the retrieval of composite components that are composed of finite numbers of available reusable components, because it is unlikely that reusable components can be provided to serve all possible functions. In such a case it is desirable for the designer to be able to think top-down, while the bottom-up search for available components is aided by the software base management system. This is necessary because the set of reusable components can get very large, so that it becomes unreasonable to expect each designer to be familiar with all of the available reusable components.

A practical approach to this problem is to consider the database to contain all the components that can be generated from a finite set of explicitly stored components by finite combinations of a set of primitive *component constructors*. Examples of component constructors are transformations that instantiate generic parameters, or that create a composite component by interconnecting a pair of available components. Retrievals from such databases will generally involve a limited degree of logical inference, to determine whether a component matching the query can be constructed from available components within a given limited number of constructor applications. Limits are needed to make sure that retrievals will always terminate. It is desirable for different limits to be specifiable as part of a query to account for differing circumstances (e.g. during a prototype demonstration session retrievals of alternative implementations must be very fast to avoid customer impatience, while during an extended design session an overnight run may be acceptable for exploring a difficult problem). These logical inferences are performed according to rules stored in the knowledge base associated with the component library.

## 5. Conclusions

Object-oriented approaches are a promising means to achieve a high level of automation in the software development process. An integrated approach is needed to realize the potentials of this approach to rapid prototyping of software systems, especially for systems with real-time constraints. The design and analysis methods, the notations and tools that support them, and the engineering databases coordinating the process must all be tied together in a consistent object-oriented style. This involves object-oriented data models at several different levels, which must be tied together by transformations and constraints. This is a promising area for research, with many interesting and tractable problems whose solutions promise great practical benefits.


1. G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1987.

2. V. Berzins and M. Gray, ''Analysis and Design in MSG.84: Formalizing Functional Specifications'', *IEEE Trans. on Software Eng. SE-11*, 8 (Aug. 1985), 657-670.

3. V. Berzins, M. Gray and D. Naumann, ''Abstraction-Based Software Development'', *Comm. of the ACM 29*, 5 (May 1986), 402-415.

4. V. Berzins, "On Merging Software Extensions", *Acta Informatica 23*, Fasc. 6 (Nov. 1986), 607-619.

5. V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1988.

6. V. Berzins and Luqi, "Languages for Specification, Design and Prototyping", in *Handbook of Computer-Aided Software Engineering*, Van Nostrand Reinhold, 1988.

7. S. Danforth and C. Tomlinson, "Type Theories and Object-Oriented Programming", *ACM Computing Surveys 20*, 1 (Mar. 1988), 29-72.

8. J. Goguen and J. Meseguer, "Unifying Functional, Object-Oriented, and Relational Programming with Logical Semantics", in *Research Directions in Object-Oriented Programming*, MIT Press, Cambridge, MA, 1987.

9. J. Guttag, Notes on Type Abstraction, Vol. SE-6, Jan. 1980.

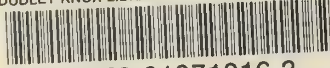
10. M. Ketabchi and V. Berzins, "Modeling and Managing CAD Databases", *IEEE Computer 20*, 2 (Feb. 1987), 93-102.

11. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Trans. on Software Eng.*, October, 1988.

12. Luqi, "Knowledge Base Support for Rapid Prototyping", *IEEE Expert*, Nov. 1988.

## Initial Distribution List

| | |
|---|---|
| Defense Technical Information Center<br>Cameron Station<br>Alexandria, VA 22314 | 2 |
| Dudley Knox Library<br>Code 0142<br>Naval Postgraduate School<br>Monterey, CA 93943 | 2 |
| Center for Naval Analyses<br>4401 Ford Avenue<br>Alexandria, VA 22302-0268 | 1 |
| Director of Research Administration<br>Code 012<br>Naval Postgraduate School<br>Monterey, CA 93943 | 1 |
| Chairman, Code 52<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, CA 93943-5100 | 1 |
| Valdis Berzins<br>Code 52Be<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, CA 93943-5100 | 100 |


DUDLEY KNOX LIBRARY - RESEARCH REPORTS
5 6853 01071016 3