NISTIR 5600

Object-Oriented Technology Research Areas

Wayne J. Salamon
Dolores R. Wallace

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

NIST

NISTIR 5600

Object-Oriented Technology Research Areas

Wayne J. Salamon
Dolores R. Wallace

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

January 1995

U.S. DEPARTMENT OF COMMERCE
Ronald H. Brown, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Arati Prabhakar, Director

ABSTRACT

Object technology is a term that covers many topics. Some of the aspects of object technology are object-oriented analysis (OOA), object-oriented design (OOD), and object-oriented programming (OOP). Also, modeling of information systems based on objects is a topic with growing interest.

This paper discusses some of the issues surrounding object technology. The topics which are discussed are object-oriented development methodologies, measuring the quality of object-oriented (OO) software, testing, the use of OO technology in high-integrity systems, and distributed object computing. The purpose of this report is to identify research topics in object-oriented technology for the NIST Computer Systems Laboratory. A bibliography is included to assist the reader in selecting material for further reading.

KEY WORDS

Distributed computing; Object-oriented design; Object-oriented programming; Object technology; Software measurement; Software testing

1.0 INTRODUCTION

Object technology is reaching a level of maturity in the software industry. The applications for object technology range from programming abstractions to database configurations to graphical user interfaces. The purpose of this report is to identify the topics and related issues in object-oriented technology that require further research to enable NIST to provide practical guidance for the users and beneficiaries of this technology.

Much of the focus in the software industry has been on using object-oriented techniques to develop software. Object-Oriented Analysis (OOA), Object-Oriented Design (OOD), and Object-Oriented Programming[1] (OOP) have been researched extensively. There are many approaches to solving problems using OOA, OOD and OOP principles. These methodologies vary in their scope and complexity. Some cover only the analysis or design aspects of a project, while others attempt to cover all phases of software development. Some methodologies are based on existing structured development techniques, while others develop new approaches to analysis and design.

Object-oriented programming garnered the most interest early on in the OO movement. Much of the discussion in the literature focused on languages for supporting OOP. As experience grew with OO techniques, the research expanded to include OOA and OOD topics. New areas of research include object-oriented domain analysis (OODA), object-oriented domain design (OODA), and object-oriented requirements analysis (OORA) [FIRESMITH].

Some of the questions to be asked are: What is different about object-oriented analysis, design, and programming, as compared to structured techniques? How and why is testing different? What are the advantages to using object-oriented techniques to develop software? What are the pitfalls?

Bertrand Meyer describes the object-oriented approach as "The emphasis on structuring a system around the classes of objects it manipulates rather than the functions it performs on them, and reusing whole data structures, together with associated operations, rather than isolated routines" [MEYER]. The key concepts are encapsulation, abstraction, and reuse. Objects are self-sufficient, and the goal is to have objects manage themselves, and provide services in an implementation neutral manner.

This paper explores some of the issues involved in software development using object-oriented techniques. The goal of the paper is to raise some questions which need further research in order to be answered. The first section discusses some of the issues involved in selecting a OO development methodology. The next section discusses measuring object-oriented software, using existing metrics as well as new metrics. Next, we describe the testing of object-oriented software, and the use of object-oriented software for high integrity systems. A section on distributed

[1]The term Object-Oriented Programming (OOP) refers to the implementation phase of software development, i.e. coding.

computing based on object models follows. The final section contains short summaries of other issues associated with object technology.

A reference section is included to provide pointers to books, papers, and articles for further information. The section is organized by topic.

2.0 OBJECT-ORIENTED DEVELOPMENT METHODOLOGIES

This section discusses some of the issues involved in comparing and choosing an OO development methodology. Work is continuing in the computer industry on OO methodologies. Many methodologies are reaching a level of maturity, resulting in more effective use of the techniques. However, some of the shortcomings of various methodologies are becoming apparent.

Most OO development methodologies have similar features. One common theme is the modeling of the problem and solution using a common modeling language in the analysis and design. One feature of OO development is that the boundary between the analysis and design is small, and often blurred. The objects used to describe the problem are often the same objects in the design. The methodologies differ in the types of models each recommends.

An overview of several OO development methodologies would be useful to combat the marketing done by many methodologists. Some of the questions raised are: What is the industry experience with the methodology? What Computer-Aided Software Engineering (CASE) tools, from the vendor and other parties, support the methodology? How effective is the methodology in modeling different software application domains, from a theoretical standpoint? Even though many of the OO methodologies have been developed from "real-world" development projects, there are still questions as to whether the techniques can be applied to other application domains. Some methodologies are specific to certain domains (Real-time Object-Oriented Methodology [ROOM]), while others claim to support a larger set of application domains.

Are some OO methodologies better suited to high integrity software, or real-time systems? The Hierarchical Object-Oriented Design (HOOD) methodology was specifically developed for large, complex, real-time distributed systems [CUTHILL]. Many of the methodology designers are from the real-time field [based on comments by Ed Berard]. Is it to our advantage to develop a methodology specifically for a single problem domain? Or are the existing methodologies sufficient to be applied to any problem domain? Many companies have learned to mix features from different methodologies an create their own OO methodologies. Examples can be found in [HEWPAC] and [LORENZ].

Much of the literature written to date has involved the theoretical aspects of applying an OO methodology to a development project. Research can be conducted on the state-of-the-practice concerning these development methodologies. A survey of companies using these methodologies can be conducted to determine in what directions industry is taking. Are companies standardizing on a methodology, or are they allowing the project managers to decide for a given project? Are companies moving to using only one methodology, or are they mixing the best attributes from different methodologies into a custom development environment? Also, what tools are out there to support software development using OO methodologies?

One issue receiving attention is the standardization of object methodologies. Many companies use a mix of object methodologies, and are interested in standards. There are various levels of standards, however. It may not be possible to derive a single object-oriented methodology

because problem domains emphasize different areas of the model. However, It may be useful to standardize the terminology, notation, and basic concepts. A discussion of standardization by six of the leading methodologists is contained in the [OOPSLA94] paper. One result of such efforts could be a standard glossary of OO terminology, or an updating to the existing software engineering glossaries such as "The IEEE Standard Glossary of Software Engineering Terminology" (ANSI/IEEE Standard 610.12). NIST can assist in producing this glossary.

3.0 OBJECT-ORIENTED SOFTWARE METRICS

The measurement of software quality is still a developing field, with many issues unresolved. There have been many metrics developed for software design and code. However, most of these metrics have been developed with structured design and programming in mind [NIST500-209]. The questions being asked now are what new metrics are needed for OO software, what existing metrics still apply (with or without modification), and the applicability of general metrics for OO software across OO methodologies. Also, there is a need to reevaluate some commonly accepted truisms about metrics. For example, for OO software, low coupling levels between classes is not always attainable due to inheritance.

Reuse is claimed to be one of the major advantages of OO software. Reuse at the code, design, and even requirements specification levels may be possible. [NISTIR5459] How does one measure reuse using existing metrics? What new metrics need to be developed? How do we measure reliability of components reused, for which the designs and source code are not available? How is productivity measured?

3.1 Existing Metrics Applied to OO Software

This section discusses a few of the more common metrics used for software development and the application of these metrics to OO software. Most software metrics are oriented towards structured programming, and therefore measure the structure of the software artifact. When dealing with OO software, which is inherently focused on the operation of the software, and less on the structure, the question arises: what role do existing metrics play in OO software?

There is much research and industry experience with software metrics. (See [IEEE982], [IEEE1045], [ISO9126], [ArmySTEP]) Some metrics have matured nearly to the point of standardization. A desire exists to use that experience and apply it to OO software artifacts. At the source code level, many of the methods developed for objects follow traditional structured programming techniques. Can we apply complexity measures to these functions? At the design level, what metrics from structured design still apply? Is it meaningful to apply a metric for module coupling to a class? Is low coupling desirable, given the benefits of inheritance for reuse? Or must we further refine the definition of coupling, to have coupling within an inheritance tree and coupling between classes in different trees? (A problem is then encountered with languages which force all classes to derive from a system root class).

Along with product metrics come process metrics. Object-oriented software is NOT typically developed using the waterfall process model. Many process metrics, however, assume a waterfall model, or a close derivative. The rapid prototyping model of development, favored by many OO developers, requires a new attitude by management. With code being changed often, and much thrown away, metrics must be applied carefully. Measuring productivity against developed code may not be wise, because so much code is not released.

Another aspect is that the design stage may not end until much of the code is complete. Designs tend to develop alongside prototype code in this development model. Some existing metrics for design count errors found in code as design errors. But these measures can be skewed by the fact that the design is meant to change when code is tested for feasibility.

3.2 New Metrics for OO Software

Researchers are now starting to define new metrics specific to OO software products [ABREU][CHIDAMBER][LI]. The three papers are discussed in this section.

In the [ABREU] paper, a framework has been defined for the metrics. This framework divides metrics into categories (design, size, complexity, reuse, productivity, and quality), subdivided by granularity. The granularity groups are method, class, and system. Table 1 summarizes the metrics. There is no experience in applying these metrics defined in [ABREU] to real-world projects. However, that effort is under way. This paper also applies some existing metrics (cyclomatic complexity, volume, information flow) to OO software.

The [CHIDAMBER] paper defines six new metrics for evaluating OO software. The metrics are evaluated against Weyuker's properties for software metrics [WEYUKER]. The metrics defined in the paper have been tested on two projects, and the results are reported in the paper. There is a need for further evaluation of the metrics because the data collected in the two experiments is sparse. The metrics are summarized in Table 2.

The final paper reviewed [LI] discusses five of the six metrics defined by Chidamber in an earlier paper than discussed above. Additional metrics are then defined, and these are summarized in Table 3. The paper also discusses prediction of maintenance effort based on the metrics data collected. The metrics were used on two commercial projects, and the paper presents several analyses of the validation results. Two models are defined for using the metrics. The first incorporates the five metrics from Chidamber, and the additional five defined by Li. The second model incorporates the two size metrics, SIZE1 and SIZE2. The results of the analyses show that there is a strong relationship between metrics and maintenance effort in OO systems.

	Method	Class	System
Design	percentage of used instance variables; ratio of comments to code size	method cohesion; depth of inheritance; number of methods available to the class	average dimension (lines of code) of methods; average number of methods per class; average number of instance variables per class
Size	number of executable statements; operator and operand counts; number of instance variables used	number of methods; total number of instance variables; size of class interface	number of classes; total number of methods; total number of instance variables
Complexity	cyclomatic complexity; volume metric; information flow	number of directly inheriting subclasses; number of subclasses that inherit directly or indirectly; number of superclasses from which the class inherits directly; number of superclasses from which the class inherits directly or indirectly	total length of inheritance chain; number of noninheritance-related couples with other classes
Reuse	internal reuse: number of times method is inherited (n1); number of times the method is overloaded (n2); method reuses efficiency (n1-n2)/n1	percentage of inherited methods that are overloaded; number of times library class is reused "as is"; number of times library class is reused with adaptation	percentage of reused "as is" classes; percentage of reused classes with adaptation; library quality factor
Productivity	effort to build an average method; new methods developed per unit effort	effort to build an average class; new classes produced per unit effort	average effort to build a class; reused classes adapted per unit effort
Quality	method reliability; number of method defects per time period; average time to identify and correct a defect	class reliability; average number of defects per method; average number of failures per method	test effectiveness; medium time between failures; average learning time

Table I: Metrics defined in the paper [ABREU]

Metric	Description
Weighted Methods Per Class (WMC)	$WMC = \sum_{i=1}^{n} c_i$ where c_i is the complexity of the method i
Depth of Inheritance Tree (DIT)	the depth of the class in the inheritance tree
Number of Children (NOC)	NOC = number of immediate subclasses subordinate to a class in the class hierarchy
Coupling between object classes (CBO)	CBO for a class is the count of the number of other classes to which it is coupled
Response For a Class (RFC)	RFC = |RS| where RS is a set of methods that can potentially be executed in response to a message received by an object of that class
Lack of Cohesion in Methods (LCOM)	LCOM = count of the number of method pairs whose similarity is 0, minus the count of method pairs whose similarity is not zero Similarity is the number of common instance variables used by two methods in a method pair

Table 2: Metrics defined in the paper [CHIDAMBER]

Metric	Description
Message-passage coupling (MPC)	MPC = number of send statements defined in a class
Data abstraction coupling (DAC)	DAC = number of ADT's defined in a class (A class is an ADT, so this metric gives the number of classes used in an aggregate class)
Number of methods in a class (NOM)	NOM = number of local methods (represents the interface complexity of the class)
Size of class (SIZE1)	SIZE1 = number of semicolons in class (LOC)
Size of class (SIZE2)	SIZE2 = number of attributes + number of local methods

Table 3: Metrics defined in the paper [LI]

4.0 TESTING OBJECT-ORIENTED SOFTWARE

Traditional software testing is usually conducted in three major phases: unit testing, integration testing, and software system testing. These test phases still occur in OO software testing, but the testing takes different approaches. We discuss the testing of OO software in these three phases in this section.

Testing of object-oriented software involves some fundamental differences with testing of procedural software. Because objects store state data, the state of the object must be determined before the test case can be run. This means that there must be some way to access the object's state during test. Debugging tools may do more harm that good, because the object under test may be affected by the tool. In [BERARD], it is argued that we must test what is delivered, which means that the complete objects must be tested, not some scaled-down form of the object which is easier to test.

Unit testing for structured software is the testing of individual software functions. The testing involves white-box testing (statement testing) and black-box testing (functionality). In OO software, unit testing is the test of classes and objects [FIRESMITH2]. The testing of objects involves more than the testing of the individual methods. Because objects store state data, the order of message processing affects subsequent object behavior. Test cases must be developed to effectively test the object with minimal number of test cases. Firesmith recommends built-in test for objects [FIRESMITH2]. One advantage to built-in test is that test case drivers do not need to have knowledge of the internal workings of the object. That knowledge would break the encapsulation of the object, and may be difficult to achieve with some languages.

Class testing is often accomplished indirectly by testing the objects instantiated from the class [FIRESMITH2]. With inheritance and generic classes, thorough testing is much more difficult. To test a generic class, test cases must be developed for each object instantiated from the class with different generic parameters. Classes which inherit attributes from ancestors require regression testing each time a ancestor class is changed. (This conflicts with the OO software promise of class encapsulation) Firesmith advocates regression testing as a normal practice, and reuse libraries should store source code, test plans and procedures, drivers, stubs, and test cases. The idea of having "stand-alone" reusable classes should extend beyond source code and libraries to include packaging of these other documents as well.

Integration testing of OO software involves the testing of messages, events, object behavior, and interfaces between the OO software and non-OO software, databases, or the operating system [FIRESMITH2][JORGENSEN]. The testing of messages includes the message passing between objects as well as the handling of the message within an object. Event testing is the testing of system functions bounded by input events and output events [JORGENSEN]. The series of messages processed to implement the functionality of the system are tested in this manner. Within this test scenario is an examination of the object processing of the messages. There must be adequate tools to allow tracing of messages as well as examining objects dynamically.

System testing of OO software is the testing of execution threads within the software system. The goal of the testing is to determine whether the assemblies of classes implement the proper system behavior. The software is tested within the complete system. Issues include software interaction with hardware devices, saving of persistent objects, and error recovery. For the latter, propagation of error conditions via messages and handling of these messages must also be tested.

Along with testing come post-mortem analysis tools. When a system crashes during test, data is often logged in order for the programmer to unwind the execution of the program leading up to the problem. Saved data include state data, execution traces, processor information, and data local to the procedure (i.e. the stack). When analyzing an OO program, the execution trace would need to include the method invocation sequence. Also, messages may need to be saved in order to recreate the scenario, as well as the state of the objects involved. Have debugging tools kept pace with the programming environments for OO software systems? With large systems, is it possible to record the state of every object involved in every test scenario? Have tools been written that only record information about objects involved in the specific test scenario that failed?

Are other approaches to post-mortem debugging required? When objects that are distributed across processors are tested, there is a need for data capture tools to communicate as well across processor domains. Or is it feasible for the objects themselves to include data capture tools as part of their operation? Performance is then a concern. Because the behavior of the system is implemented by the object interactions, what information must be saved during system test in order to recreate test scenarios, without executing the entire test suite from the start? These problems are not new to OO software testing, but analyzing the system states requires knowledge of object activity.

5.0 OBJECT-ORIENTED SOFTWARE FOR HIGH INTEGRITY SYSTEMS

High integrity software is software that must be trusted to work dependably in some critical function, and whose failure to do so may result in serious injury, loss of life or property, business failure or breach of security [NIST500-204]. Many standards and guidelines have been produced for software used in high integrity systems. How does the introduction of OO technology affect these standards? Are the standards too dependent on structured design methods and associated lifecycles?

The choice of what programming language to use for specific types of systems has been debated since the early days of the computer industry. Tradeoffs of speed versus the abstraction supported by the language have been discussed extensively. With OO programming languages, what are the new issues raised, and what existing issues are recast in a new light? For example, how does the run-time system for managing objects affect the behavior of the program? Object construction, storage allocation, destruction, and garbage collection may be out of control of the programmer. In the C++ language, default object constructors and destructors are called when objects are created and destroyed, but these default constructors and destructors may be redefined by the programmer. Having the programmer code the constructors and destructors is necessary to ensure that the behavior of the program can be analyzed statically by looking at the code. However, it is better to have the constructors and destructors specified in the design.

How does the use of OO technology affect the design of high-integrity systems? The paper by Cuthill discusses four principles of good software design and the relationship of OO design to these principles [CUTHILL]. These principles are modularity, functional diversity, traceability, and removal of ambiguity. Modularity is the dividing of the system into logically separate components with defined interactions and limited access to data. Functional diversity provides for multiple independent checks on the data produced. Traceability is the ease of mapping requirements, analysis, design, and code to each other. Removal of ambiguity means having no code with unpredictable effects.

The conclusions in the [CUTHILL] paper are now summarized. OO design supports modularity well through abstraction, although care must be taken to reduce the complexity of inheritance trees. Because OO design is still evolving, many methods do not address the issue of managing a large number of classes. One example of a methodology that does address the issue is in [FIRESMITH]. The subassembly, assembly, and framework models provide a method to group together classes that are related. Other methodologies provide similar grouping techniques based on different rationales [CUTHILL].

Functional diversity is supported by encapsulation, with independent objects providing similar functions and communicating to coordinate the results. There is a danger with having the independent objects connected via the inheritance tree. Care must be taken to design the classes to reduce or eliminate any dependencies between the objects.

OO design supports traceability between the analysis, design, and coding via the object models. Because the objects described in the analysis and design are the same objects in the code, traceability is enhanced. One problem is that many methodologies do not support all phases of the lifecycle (analysis, design, and code). However, some methodologies are being updated or developed which do support more than one phase. Examples include [BOOCH94] and [FIRESMITH].

Removal of ambiguity is not a problem in the OO design phase [CUTHILL]. The Cuthill paper discusses the ambiguity problem in relation to C++ in addition to the other three principles. A similar analysis of other OO programming languages may be useful. Also, what features of the languages support other OO design goals in addition to the four principles discussed? A paper by SoHaR Inc. [SOHAR] discusses C, C++, Ada, and PL/M in relation to OO design goals such as abstraction, encapsulation, modularity, hierarchy, and typing, in addition to concurrency and persistence. A similar study with Ada95, C++ (because the language has changed since the SoHaR paper was written), Eiffel, and Smalltalk could be conducted.

There has been research conducted in distributed, real-time systems based on object models. One paper by Takashio and Tokoro describes distributed real-time object model and programming language [TAKASHIO]. Another paper by Satoh and Tokoro introduces a formal model and language for reasoning about real-time object-oriented computations [SATOH]. Is there a similar formal model for describing high integrity software? Can we capture the semantics of dependable object systems in some formal language?

6.0 DISTRIBUTED OBJECT COMPUTING

One technology gaining considerable support is distributed computing based on an object model. The application program becomes a loosely coupled system of components. These components may reside within one process, one computer, one network, or across different networks. The components are modeled within the systems as large-grained objects. There are several enabling technologies to allow these components to communicate. Examples include Common Object Request Broker Architecture (CORBA) from the Object Management Group, Object Linking and Embedding and Common Object Model (OLE/COM) from Microsoft, Distributed Objects Everywhere (DOE) from Sunsoft, and System Object Model (SOM) from IBM. Comparing and contrasting these technologies with Remote Procedure Call (RPC) and the Distributed Computing Environment (DCE) is useful to understand different approaches to distributed computing.

The enabling technologies provide different levels of abstraction for component modeling. Remote Procedure Call, for example, provides a point-to-point mechanism for communication between processes. This form of communication fits well with the process model. The CORBA model provides for indirect communication between objects, with the object request broker providing the message services between objects. The requesting object does not need to know where, or even if, the target object is active. The broker will locate the object, or activate the object if needed. Object modeling provides a consistent design for these types of applications.

In comparing these different enabling technologies, we must decide what criteria we will use to make the comparison. Complexity, portability, abstraction, and performance are all valid areas to investigate. The RPC mechanism may be too low-level at the application level to be used in a system with hundreds of objects. Also, RPC increases the coupling between these objects. A CORBA product may reduce the inter-object dependencies, but at what cost to performance and portability?

We can move up to another level of abstraction, away from object-to-object communication altogether. The idea of a "software bus" has been around for many years. Can a system be developed where objects place information on an underlying communication system, without sending messages directly to other objects? How do the target objects know what information to retrieve from the bus? The advantages of such an architecture is eliminating the communication problem where an object must understand the messages of all objects it communicates with. Some OO frameworks are heading in this direction. For example, MITRE's DISCUS [DISCUS] framework provides object communication via pre-defined services. Frameworks are discussed further is section 7.0.

7.0 OTHER ISSUES

One of the main features of OO software is the encapsulation of data and operations. One of the claims is that changes to an OO software system are more localized and side-effects are minimized. However, there are still potential problems which can arise. For example, if a base class, high up in the inheritance tree, is changed, many modules may need to be recompiled. This recompilation may be necessary even though the descendent classes do not take advantage of the change. One issue is then, how are components going to be updated in the system without requiring access to source code of the changed component? There are several products which address this issue. IBM's System Object Model is one architecture designed to isolate changes.

A related issue is the effect on object behavior created by a change to another object. Can it be guaranteed that changes to an object's internal behavior will not effect dependent objects? An example is an object which changes from ignoring bad message operands to raising exceptions. What effect will this new behavior have on dependent objects, and overall system behavior? Even though the object's interface did not change, how other objects interact with this object can not be ignored. Also, what effects will the change to the class have on subclasses of that class, in terms of the subclasses' behaviors? Are the subclasses necessarily improved by the change, or are potential problems with the subclasses going to be surfaced? How much regression testing is then needed of all subclasses? (This problem is sometimes called the "fragile base-class" problem).

The choice of which language to use to implement object-oriented software requires careful consideration. C++ is the most popular language, but Smalltalk and Eiffel have substantial followings. There are many issues involved in comparing languages. For example, the issue of class dependencies on base classes. Every class in Smalltalk is a descendent of a single root class; there is only one inheritance tree in a Smalltalk program. C++, however, allows for multiple inheritance trees, independent of each other. Which language is "better" for implementing the model of the problem?

The idea of object frameworks is a growing area of interest within the object community. These frameworks provide an object model for application development within a specific problem domain. The SEMATECH [SEMATECH] framework for computer-integrated manufacturing (CIM) of semiconductors, and the DISCUS framework mentioned above are two examples. Other companies are developing frameworks for the health care industry. One possible area of research is to abstract the common features of these frameworks into a reference model for developing frameworks for other domains.

Frameworks have also been developed for specific types of application functions. Graphical user interfaces (GUI) are the most common types of these frameworks. The Taligent [TALIGENT] system includes frameworks for GUIs, file access, and other functions common across applications. Also, the Taligent framework is extensible so domain specific frameworks can be developed and integrated into the Taligent system. The Taligent framework builds on top of the System Object Models (SOM and Distributed SOM) developed by IBM.

Other examples of frameworks include Borland's ObjectWindows Library and Microsoft Foundation Classes. These frameworks provide a model of the graphical user interface in the Microsoft Windows environment. Instead of programming individual windows and their components, the programmer creates instances of window, menu, and dialog box objects for the application.

8.0 SUMMARY

The purpose of this report is to identify the topics and related issues in object-oriented technology that require further research to enable NIST to provide practical guidance for the users and beneficiaries of this technology.

Research is needed on the following topics to help clarify issues in software engineering that will enable high integrity software systems, which may be produced economically:

- characteristics for deciding when to use OO technology for various application domains
- measurement of quality of OO programs
- measurement of productivity of using OO technology
- testing of OO programs
- use of OO to facilitate software reuse
- use of OO in safety-critical systems
- use of OO for distributed computing systems
- use of OO with existing traditional systems
- transferring existing software engineering knowledge and experience to OO technology
- standardization of OO terminology.

Research on these topics should be developed into practical guidance for those who develop, assure, purchase, use, or license high integrity software systems.

9.0 REFERENCES

[ABREU]

Abreu, Fernando Brito e and Rogerio Carapuca, "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework," *Journal of Systems Software*, no. 26 (1994).

[ArmySTEP]

Betz, Henry P. and Patrick J. O'Neill, "Software Metrics Initiatives Report," Army Software Test and Evaluation Panel (STEP), March 21, 1991.

[BERARD]

Berard, Edward, Testing of Object-Oriented Software, (Course workbook), Berard Software Engineering, Inc., 1994.

[BOOCH94]

Booch, Grady, Objected-Oriented Analysis and Design with Applications, second edition, The Benjamin/Cummings Publishing Company, Inc., 1994.

[CHIDAMBER]

Chidamber, Shyam R. and Chris F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6 (June 1994).

[CUTHILL]

Cuthill, Barbara B., "Applicability of Object-Oriented Design Methods and C++ to Safety-Critical Systems," *Proceedings of the Digital Systems Reliability and Nuclear Safety Workshop*, NIST Special Publication 500-216, March 1994.

[DISCUS]

Data Integration and Synergistic Collateral Usage Study (DISCUS), (Tutorial workbook), The MITRE Corporation, November 2, 1994.

[FIRESMITH]

Firesmith, Donald G., Object-Oriented Requirements Analysis and Logical Design, John Wiley & Sons, Inc., 1993.

[FIRESMITH2]

Firesmith, Donald G., "Testing Object-Oriented Software," whitepaper, August 12, 1994. Also appears in *Software Engineering Strategies*, Auerbach Publications, Vol. I, No. 5, (Nov./Dec. 1993).

[HEWPAC]

de Champeaux, Dennis, Douglas Lea and Penelope Faure, Object-Oriented System Development, Prentice-Hall, 1993.

[JORGENSEN]
Jorgensen, Paul C. and Carl Erickson, "Object-Oriented Integration Testing," *Communications of the ACM*, Vol. 37, No. 9 (September, 1994).

[IEEE982]
IEEE Std. 982.1-1988, "IEEE Standard Dictionary of Measures to Produce Reliable Software," The Institute of Electrical and Electronics Engineers, June, 1988.

[IEEE1045]
IEEE Std. 1045-1993, "Standard for Software Productivity Metrics," The Institute of Electrical and Electronics Engineers.

[ISO9126]
ISO/IEC 9126, "Information technology - Software product evaluation - Quality characteristics and guidelines for their use," International Organization for Standardization and International Electrotechnical Commission, December 1991.

[LI]
Li, Wei and Sallie Henry, "Object-Oriented Metrics that Predict Maintainability," *Journal of Systems Software*, no. 23 (1993).

[LORENZ]
Lorenz, Mark, Object-Oriented Software Development: A Practical Guide, Prentice-Hall, 1993.

[MEYER]
Meyer, Bertrand and P. Hucklesby, "EIFFEL: an introduction," Object-oriented Programming Systems: Tools and Applications, edited by J. J. Florentin,Chapman & Hall, 1991.

[NIST500-204]
Wallace, Dolores R., Laura M. Ippolito and D.R. Kuhn, NIST Special Publication 500-204, "High Integrity Software Standards and Guidelines," U.S. Department of Commerce/National Institute of Standards and Technology, prepared for U.S. Nuclear Regulatory Commission, September 1992.

[NIST500-209]
Peng, Wendy W. and Dolores R. Wallace, NIST Special Publication 500-209, "Software Error Analysis," U.S. Department of Commerce/National Institute of Standards and Technology, April 1993.

[NISTIR5459]
Salamon, W. J. and D. R. Wallace, NIST Internal Report 5459, "Quality Characteristics and Metrics for Reusable Software (Preliminary Report)," U.S. Department of Commerce/National Institute of Standards and Technology, May 1994.

[OOPSLA94]
Monarchi, David, Grady Booch, Brian Henderson-Sellers, Ivar Jacobson, Steve Mellor, James Rumbaugh and Rebecca Wirfs-Brock, "Methodology Standards: Help or Hindrance?" *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vol. 29, No. 10, October 1994.

[RUMBAUGH]
Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, Object-Oriented Modeling and Design, Prentice Hall, 1991.

[SATOH]
Satoh, Ichiro and Mario Tokoro, "A Formalism for Real-Time Concurrent Object-Oriented Computing," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vol. 27, No. 10, October 1992.

[SEMATECH]
Computer Integrated Manufacturing (CIM) Application Framework, SEMATECH, Inc., March 31, 1994.

[SOHAR]
Tai, Ann and Herbert Hecht, "A Comparative Study of Programming Languages for Class 1E Applications," SoHaR Incorporated, Beverly Hills, CA, September 1991.

[TAKASHIO]
Takashio, Kazunori and Mario Tokoro, "DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vol. 27, No. 10, October 1992.

[TALIGENT]
"Leveraging Object-Oriented Frameworks," Taligent, Inc., 1993.

[WEYUKER]
Weyuker, Elaine J., "Evaluating Software Complexity Measures," *IEEE Transactions on Software Engineering*, vol. 14, no. 9 (September 1988).

10.0 GLOSSARY

ancestor class — A class that cannot have direct instances but whose descendants can have instances [RUMBAUGH]

binding — Denotes the association of a name (such as a variable declaration) with a class [BOOCH94].

class — A set of objects that share a common structure and a common behavior [BOOCH94];
a template from which objects can be instantiated [FIRESMITH];
a description of a group of objects with similar properties, common behavior, common relationships, and common semantics [RUMBAUGH].

dynamic binding — A binding in which the name/class association is not made until the object designated by the name is created at execution time [BOOCH94].

inheritance — A relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes; [BOOCH94];
a relationship among classes that allows subclasses to be built as extensions or specializations of superclasses [FIRESMITH].

instance — Something you can do things to; has state, behavior, and identity; the terms *instance* and *object* are interchangeable [BOOCH94];
an object constructed by instantiation from a class [FIRESMITH];
an object described by a class [RUMBAUGH].

instantiate — To create a new instance of a class or type [FIRESMITH].

instantiation — The process of filling in the template of a generic or parameterized class to produce a class from which on can create instances [BOOCH94];
the process of creating instances from classes [RUMBAUGH].

link — Between two objects, one instance of an association [BOOCH94];
an instance of an association; a physical or conceptual connection between objects [RUMBAUGH].

message

An operation that one object performs upon another; the terms *message, method,* and *operation* are usually interchangeable [BOOCH94];
the primary means of communication among classes and objects consisting of a request for service, a notification of an event, or the passing of data [FIRESMITH];
(in Smalltalk) invocation of an operation on an object, comprising an operation name and a list of argument values [RUMBAUGH].

method

An operation upon an object, defined as part of the declaration of a class; all methods are operations, but not all operations are methods [BOOCH94];
a popular synonym for operation [FIRESMITH];
the implementation of an operation for a specific class [RUMBAUGH].

object

Something you can do things to; has state, behavior, and identity; the terms *instance* and *object* are interchangeable [BOOCH94];
an abstraction (i.e. model) in the requirements, design, and/or code of a single tangible or intangible object, entity, or thing from the real-world, application, or problem domain [FIRESMITH];
a concept, abstraction, or thing with crisp boundaries and meanings for the problem at hand; and instance of a class [RUMBAUGH].

operation

Some work that one object performs upon another in order to elicit a reaction [BOOCH94];
a discrete activity, action, or behavior that implements a functional (i.e. sequential) or process (i.e. concurrent) abstraction, and that is typically performed by, belongs to, and is part of an object or class [FIRESMITH];
a function or transformation that may be applied to objects in a class [RUMBAUGH].

polymorphism

A concept in type theory, according to which a name may denote objects of many different classes that are related by some common superclass [BOOCH94];
the ability of the same identifier to refer at run-time to different instances of various classes [FIRESMITH];
takes on many forms; the property that an operation may behave differently on different classes [RUMBAUGH].

state

The cumulative results of the behavior of an object; one of the possible conditions in which an object may exist, characterized by definite quantities that are distinct from other quantities [BOOCH94];
the values of the attributes and links of an object at a particular time [RUMBAUGH].

static binding A binding in which the name/class association is made when the name is declared (at compile time) but before the creation of the object that the name designates [BOOCH94].

subclass A class that inherits from one or more classes (which is called its immediate *superclasses*) [BOOCH94];
a refined version of another class, the superclass [RUMBAUGH].

superclass The class from which another class inherits (which is called its immediate *subclass*) [BOOCH94];
a more abstract version of another class, the subclass [RUMBAUGH].

11.0 BIBLIOGRAPHY

11.1 Methodologies

Booch, Grady, Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc., 1983.

Booch, Grady, Objected-Oriented Analysis and Design with Applications, second edition, The Benjamin/Cummings Publishing Company, Inc., 1994.

Booch, Grady, "Object-Oriented Development," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2 (February 1986).

Coad, Peter and Edward Yourdon, Object-Oriented Analysis, second edition, Yourdon Press, 1991.

Coad, Peter and J. Nicola, Object-Oriented Programming, Yourdon Press, 1993.

de Champeaux, Dennis, Douglas Lea and Penelope Faure, Object-Oriented System Development, Prentice-Hall, 1993.

Fichman, Robert G. and Chris F. Kemerer, "Object-Oriented and Conventional Analysis and Design Methodologies," *IEEE Computer*, October 1992.

Firesmith, Donald G., Object-Oriented Requirements Analysis and Logical Design, John Wiley & Sons, Inc., 1993.

Henderson-Sellers, Brian, and Julian M. Edwards, "The Object-Oriented Systems Life Cycle," *Communications of the ACM*, vol. 33, no. 9 (September 1990).

HOOD Working Group, HOOD Reference Manual, (HRM/91/07/V3.1), European Space Agency, 1991.

Korson, Tim, and John D. McGregor, "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, vol. 33, no. 9 (September 1990).

Meyer, Bertrand, Object-Oriented Software Construction, Prentice Hall, 1988.

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, Object-Oriented Modeling and Design, Prentice Hall, 1991.

Shlaer, Sally and Steven J. Mellor, Object-Oriented Systems Analysis, Yourdon Press, 1988.

Wirfs-Brock, Rebecca, Brian Wilkerson and Lauren Wiener, Designing Object-Oriented Software, Prentice Hall, 1990.

11.2 Testing

Berard, Edward, Testing of Object-Oriented Software, (Course workbook), Berard Software Engineering, Inc., 1994.

Chung, Chi-Ming and Ming-Chi Lee, "Object-Oriented Programming Testing Methodology," *Proceedings of the Fourth International Conference on Software Engineering and Knowledge Engineering*, IEEE, 1992.

Firesmith, Donald G., "Testing Object-Oriented Software," whitepaper, August 12, 1994. Also appears in *Software Engineering Strategies*, Auerbach Publications, Vol. I, No. 5, (Nov./Dec. 1993).

Jorgensen, Paul C. and Carl Erickson, "Object-Oriented Integration Testing," *Communications of the ACM*, Vol. 37, No. 9 (September, 1994).

McCabe, Thomas J., Lori A. Dreyer, Albert J. Dunn and Arthur H. Watson, "Testing an Object-Oriented Application," *CASE Outlook*, 1994.

Parrish, Allen S., Richard B. Borie and David W. Cordes, "Automated Flow Graph-Based Testing of Object-Oriented Software Modules," *Journal of Systems Software*, no. 23 (1993).

11.3 Metrics

Abreu, Fernando Brito e and Rogerio Carapuca, "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework," *Journal of Systems Software*, no. 26 (1994).

Chidamber, Shyam R. and Chris F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6 (June 1994).

Li, Wei and Sallie Henry, "Object-Oriented Metrics that Predict Maintainability," *Journal of Systems Software*, no. 23 (1993).

Rajaraman, Chandrashekar and Michael R. Lyu, "Reliability and Maintainability Related Software Coupling Metrics in C++ Programs," *Proceedings of the Third International Symposium on Software Reliability Engineering*, IEEE, 1992.

11.4 Frameworks

Data Integration and Synergistic Collateral Usage Study (DISCUS), (Tutorial workbook), The MITRE Corporation, November 2, 1994.

Computer Integrated Manufacturing (CIM) Application Framework, SEMATECH, Inc., March 31, 1994.

"Leveraging Object-Oriented Frameworks," Taligent, Inc., 1993.