IEEE Guide to Software Design Descriptions

Circuits and Devices

Communications Technology

IEEE Computer Society

Sponsored by the
Software Engineering Standards Subcommittee of the
Technical Committee on Software Engineering

Electromagnetics and Radiation

Energy and Power

Industrial Applications

Signals and Applications

Standards Coordinating Committees

IEEE Std 1016.1-1993

Published by the Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA.

May 25, 1993 *SH16071*

THIS PAGE WAS
BLANK IN THE ORIGINAL

IEEE Guide to Software Design Descriptions

Sponsor

Software Engineering Standards Subcommittee
of the
Technical Committee on Software Engineering
of the
IEEE Computer Society

Approved March 18, 1993

IEEE Standards Board

Abstract: The application of design methods and design documentation recommended in IEEE Std 1016-1987 is described. Several common design methods are used to illustrate the application of IEEE Std 1016-1987, thus making the concepts of that standard more concrete. The information in this guide may be applied to commercial, scientific, or military software that runs on any computer. Applicability is not restricted by the size, complexity, or criticality of the software.
Keywords: design entity, design method, design view, software design process

---

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

ISBN 1-55937-297-4

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Introduction

(This introduction is not a part of IEEE Std 1016.1-1993, IEEE Guide to Software Design Descriptions.)

Audience

This guide is intended for those in technical and managerial positions who prepare and use software design descriptions. It will guide in understanding IEEE Std 1016-1987 and in the selection, organization, and presentation of design information. This guide will also help standards developers to apply IEEE Std 1016-1987 in preparing design documentation standards.

At the time this standard was completed, the Software Design Description Working Group had the following membership:

Basil Sherlund, *Chair*

Don Chandler	John McArdle	Roger Van Scoy
Francois Coallier	Diane Mularz	John Wilson
Pat Finnigan	Ilona Rossman	Rodger Zambs
	David Spence	

In the final stage of revision, responses to ballot comments was provided by:

Basil Sherlund, *Chair*

William Hefley	Vladan Jovanovic

The following persons were on the balloting committee that approved this document for submission to the IEEE Standards Board:

G. Arato	H. Harauz	S. R. Schach
D. Avery	W. Hefley	G. D. Schumacher
J. H. Barnard	J. W. Horch	R. W. Shillato
R. Berlack	M. S. Karasik	D. M. Siefert
F. Buckley	R. Kosinski	H. M. Sneed
G. Cozens	T. Kurihara	A. Sorkowitz
B. K. Derganc	R. Lane	V. Srivastava
E. Dragstedt	B. Lau	R. H. Thayer
J. W. Fendrich	B. Livson	G. D. Tice
K. Fortenberry	J. Maayan	L. L. Tripp
J. Garbajosa	J. Marijarvi	R. Van Scoy
D. Gelperin	R. Martin	D. Wallace
Y. Gershkovitch	S. C. Mathews	W. M. Walsh
J. Gonzalez	I. Mazzo	P. R. Work
D. A. Gustafson	S. Mcgrath	J. Zalweski
	D. E. Nickle	

When the IEEE Standards Board approved this standard on March 18, 1993, it had the following membership:

Wallace S. Read, *Chair* **Donald C. Loughry,** *Vice Chair*
Andrew G. Salem, *Secretary*

Gilles A. Baril
Clyde R. Camp
Donald C. Fleckenstein
Jay Forster*
David F. Franklin
Ramiro Garcia
Donald N. Heirman
Jim Isaak
Ben C. Johnson
Walter J. Karplus
Lorraine C. Kevra
E. G. "Al" Kiener
Ivor N. Knight
Joseph L. Koepfinger*
D. N. "Jim" Logothetis
Don T. Michael*
Marco W. Migliaro
L. John Rankine
Arthur K. Reilly
Ronald H. Reimer
Gary S. Robinson
Leonard L. Tripp
Donald W. Zipse

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal
James Beall
Richard B. Engelman
David E. Soffrin
Stanley Warshaw

Rachel Auslander
IEEE Standards Project Editor

Contents

CLAUSE PAGE

1. Overview 1

1.1 Purpose 1
1.2 Scope 1

2. References 1

3. Definitions 2

4. Description of IEEE Std 1016-1987 3

5. Design description organization 4

5.1 Design views 4
5.2 Recommended design views 4
5.3 Design description media 5

6. Considerations 6

6.1 Selecting representative design methods 6
6.2 Representative design method descriptions 6
6.3 Design document sections 7
6.4 Method-oriented design documents 7

7. Design methods 8

7.1 Function-oriented design methods 8
7.2 Data-oriented design methods 9
7.3 Real-time control-oriented design methods 10
7.4 Object-oriented design methods 11
7.5 Formal language-oriented design methods 13

8. Bibliography 15

ANNEXES

A.1 Data flow diagrams (DFDs) 16
A.2 Transformation graphs (TRGs) 16
A.3 State transition diagrams (STDs) 17
A.4 Structure charts (STCs) 17
A.5 Entity relationship diagrams (ERDs) 18
A.6 Objects 19

IEEE Guide to Software Design Descriptions

1. Overview

This guide is divided into eight clauses and has one annex. Clause 1 provides the purpose and scope of this guide. Clause 2 lists references to other standards that are useful to applying this guide. Clause 3 provides definitions of terms used in this guide. Clause 4 briefly describes IEEE Std 1016-1987. Clause 5 provides the context for this guide. Clause 6 provides the considerations regarding this guide and introduces the concept of a design description. Clause 7 provides a description of common design methods and their relationships to IEEE Std 1016-1987. Clause 8 contains a bibliography. The annex, although not part of this guide, provides a basis for selecting a design method.

1.1 Purpose

The purpose of this guide is to show how to apply IEEE Std 1016-1987, IEEE Recommended Practice for Software Design Descriptions. The approach used to do this is as follows: The relationship between IEEE Std 1016-1987 and design methods that are familiar and widely used is shown. This includes identifying what design entities each method recognizes and what entity attributes it emphasizes. The set of models (or views) that each design method requires are also identified.

1.2 Scope

This guide describes the application of design methods and design documentation recommended in IEEE Std 1016-1987. Since most design methods specify media and information, organization of the material is left up to the user of the design method. This guide shows how the information requirements of IEEE Std 1016-1987 are met by several common design methods. IEEE Std 1016-1987 is not limited to the specific methods described in this guide. Rather, these examples were selected to illustrate the application of IEEE Std 1016-1987. The methods selected as examples are widely used and are familiar, thus making the concepts of IEEE Std 1016-1987 more concrete.

This information in this guide may be applied to commercial, scientific, or military software that runs on any computer. Applicability is not restricted by the size, complexity, or criticality of the software.

2. References

This guide shall be used in conjunction with the following publications:

IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology (ANSI).[1]

IEEE Std 1016-1987, IEEE Recommended Practice for Software Design Descriptions (ANSI).

[1]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

3. Definitions

This clause contains key terms as they are used in this guide. Definitions of other terms used in this guide can be found in IEEE Std 610.12-1990.[2]

3.1 component: A distinct part of a subsystem. A component may be decomposed into other components and computer software units.

3.2 design description: A document that describes the design of a system or component. Typical contents include system or component architecture, control logic, data structures, input/output formats, interface descriptions, and algorithms.

3.3 design entity: An element (component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.

3.4 design method: A definition of a set of essential entities. It includes a systematic procedure for the set of models with rules or heuristics used to determine the entities and entity attributes of the models, and a notation to represent the entities and the attributes expressed in each model.

3.5 design methodology: A guideline identifying how to design software. As a process, a methodology is a practical set of procedures that facilitate the design of software.

3.6 design view: A subset of design entity attribute information that is specifically suited to the needs of a software project activity.

3.7 entity attribute: A named characteristic or property of a design entity that provides a systematic procedure for the statement of fact about the entity.

3.8 model: A representation of one or more aspects of a system.

3.9 notation: A set of symbols used to represent design entities and entity attributes.

3.10 software configuration item, or subsystem: A collection of software elements treated as a unit for the purpose of configuration management.

3.11 software design description (SDD): A representation of a software system created to facilitate analysis, planning, implementation, and decision making. It is a blueprint or model of the software system. The SDD is used as the primary medium for communicating software design information.

3.12 software design document: The output of design process in a presentable format, traditionally, a paper-based document.

3.13 software design process: Organized tasks and activities of design, having appropriate specification.

3.14 software design process specification: Know-how, technology of design, that specify operationally how to use methodology of design (standardized itself) together with standards for evaluating design, tools to support design automation, and documentation required to represent design information.

[2]Information on references can be found in clause 2.

4. Description of IEEE Std 1016-1987

IEEE Std 1016-1987 is an information standard (that is, it specifies the information that a design method should contain), rather than a documentation standard (that is, a standard that specifies format). It prescribes the minimum information requirements that must be met by a design description, but it does not prescribe an organization for that information in a document.

Section 5 of IEEE Std 1016-1987 introduces the notion of a design entity and indicates the minimum set of design entity attributes that shall be specified in a design description. These attributes include the following:

a) Identification
b) Type
c) Purpose
d) Function
e) Subordinates
f) Dependencies
g) Interface
h) Resources
i) Processing
j) Data

Section 6 of IEEE Std 1016-1987 introduces the notion of a design view. A design view is a subset of design entity attribute information that is specifically suited to the needs of a software project activity. Design views may provide subsets of design entity attribute information such as hierarchy, control flow, data flow, data structure, time dependent behavior, or mode/state transition control. Partial design descriptions, such as those presented in sections of a design document, are design views. They describe something about the system but not the entire design. IEEE Std 1016-1987 provides a set of four common views. They are the following:

a) Decomposition
b) Dependency
c) Interface
d) Detailed design

These views are not required to meet the recommended practices in IEEE Std 1016-1987. Rather, they are introduced to illustrate the concept of views. Different design methods will usually have a different set of design views that are considered important.

5. Design description organization

This clause discusses the organization of information into design views, recommended design views, and the various media that can be used to illustrate a design description.

5.1 Design views

A design view is a means of organizing design information. It is a subset of entity attribute information. A complete design description provides all attribute information about all entities. However, a design view looks at only the following:

a) *A subset of the design entities.* In this situation the design view gives all design information about part of the system.
b) *A subset of the design attributes.* In this situation the design view gives a certain type of information about the entire system.
c) *Both the design entities and design attributes subsets.* In this situation the design view gives a certain type of information about part of the system.

Organization of information into design views is important for several reasons.

a) It allows a specific user of design information to focus on the information that the user considers important.
b) It decreases the scope, and thus the complexity, of information that is presented at any one time.
c) It makes it easier to consider some information globally, without it being clouded with extraneous information.

5.2 Recommended design views

IEEE Std 1016-1987 requires the organization of information into design views. However, it does not require any particular design views. It introduces a set of four separate design views, which together constitute a complete design description. These views are:

a) Decomposition description
b) Dependency description
c) Interface description
d) Detail description

These four views partition the design by considering only a subset of the entity attributes in each view. IEEE Std 1016-1987 provides descriptive information indicating the use of each view in terms of how it satisfies the information needs of some class of users.

These four design views are not necessarily the optimal views for every software project. It is possible that a class of users will be interested in a different subset of information than is presented in any one of these views. In addition, these views are sometimes not consistent with the views prescribed by design methods used on the project (see clause 7.). For example, data flow diagrams can be used to show both subordinate relationships (explosions of processes) and dependencies (data flows). In the four views above, these two attributes are represented in different views.

Designs may also be partitioned by subsets of entities without conflicting with the set of design views listed in IEEE Std 1016-1987. This is the case when design documentation is ordered by component or structure of the software.

Consequently, IEEE Std 1016-1987 should not be interpreted as requiring the four design views it introduces in its Section 6. These four design views provide a useful set of views in the absence of other considerations. The requirement is merely that the design be partitioned into design views that meet the specific information needs of different classes of users. The needs should be evaluated for each project based on design methods, tools, and media used.

5.3 Design description media

IEEE Std 1016-1987 does not prescribe or require any particular representation media. Design descriptions may be contained in paper design documentation, design languages, CASE tool dictionaries, or other formats. However, as the standard points out, paper documentation is currently the primary design description medium. For that reason, and because this guide is itself represented in a paper medium, some of the concepts of design description organization can be best illustrated using this medium.

6. Considerations

This clause discusses the criteria used in selecting representative design methods and their descriptions, design document sections, and the selection of method-oriented design documents.

6.1 Selecting representative design methods

This subclause introduces the concept of a method description. Various design methods are described further in clause 7. Numerous software design methods exist and continue to evolve as the field of software engineering evolves. This guide in no way attempts to be all-inclusive for all instances of design methods available. Rather, examples of design approaches are presented to illustrate how the various representations used in that design method capture some subset of information required by IEEE Std 1016-1987. The following broad classes of design methods are recognized:

a) Function-oriented
b) Data-oriented
c) Real-time control-oriented
d) Object-oriented
e) Formal language-oriented

This list is not meant to be exhaustive. These classes were selected because they serve to broadly categorize a large number of available methods. Specific instances of them are selected as representative of the class to show, through example, a wide spectrum of methods and their relationship to IEEE Std 1016-1987. The approach taken in this guide may be applied to a particular design method to determine where the required information is (or is not) captured.

A representative instance of each class of design method is presented relative to information recommendations of IEEE Std 1016-1987. The particular method chosen for each class was selected based on the following criteria:

a) The method conforms to the definition of a design method as employed in this guide. (That is, it defines a notation, a set of required design views, and a set of heuristics for production of the design.)
b) There is a definitive, openly published reference that describes the specific method.
c) The method has received acceptance and actual use in the community, giving it broad appeal and familiarity.

These criteria are essential to achieve the purpose of this guide. Without conformance to the criteria, clarification of IEEE Std 1016-1987 cannot be achieved. Because object-oriented methods are continuing to evolve, these criteria could not be completely satisfied. However, this class of methods was deemed to be important to describe. The selected method supports the essential concepts of this class of methods and comes close to meeting the selection criteria.

6.2 Representative design method descriptions

Each representative design method is described in a separate section of this guide. A detailed description of the method is not included in this guide. For more specific information, a bibliography is provided in clause 8. The method is only explained at the level of detail required to explain the relationships to IEEE Std 1016-1987. Each design method description presents the following information:

a) A general description of that class of design method
b) A brief overview of the representative method chosen to include a key reference citation

c) An identification of the entities and attributes captured by the method thereby providing a relationship to IEEE Std 1016-1987 (This relationship is done via the design views of the particular method. For each design view employed in the method, the relevant entities and attributes captured in the view are identified.)

6.3 Design document sections

Design documents are a means of representing design information. Each section or subsection of the design document presents a subset of information about the design of the software. Thus, sections of the design document are themselves design views.

Generally, design document standards prescribe a table of contents for the document and give narrative descriptions of the information that goes into each section. Since most design methods specify media and information, organization of the material is left up to the user of the design method. In the terms of IEEE Std 1016-1987, each section of the design document is a design view and the narrative descriptions indicate what subset of entity/attribute information should be described in that view.

In theory, the information has been partitioned into document sections that suit the needs of specific users. One way of focusing attention for different users is to put the information that each user considers important into different sections of the design document. However, the following are other considerations for the partitioning of information in design documents:

a) The design description may be broken up into two or more documents that correspond to the design development process phase, e.g., an architectural design document and a detailed design document.
b) The design document sections may correspond to the models (design views) identified in the design method used on the project.
c) The design document sections may correspond to the structure of the software, e.g., there may be a section for each major component of the software.
d) The design document sections may correspond to the different user classes, e.g.,
    1) An interface section for programmers who must write software that uses the functions of this software
    2) A quality evaluation section for quality assurance personnel
    3) An identification section for configuration managers
    d) A detailed design section for those who will program the software for this design, and so on.

Most design document standards are a hybrid of the above approaches, which results in a design document that presents a complete description, and still partitions design views in a way to effectively convey information to different classes of users.

IEEE Std 1016-1987 provides an appendix showing a table of contents for a design document based on the four primary design views it has identified. This table of contents is not mandatory, and alternative standards such as DI-MCCR-80012A (see [B4][3]) are fully compatible with IEEE Std 1016-1987.

6.4 Method-oriented design documents

Clause 7 of this guide clarifies IEEE Std 1016-1987 by projecting the terminology and concepts of that standard onto the terminology and concepts of familiar and widely used design methods. Method-oriented design documents were selected because other approaches could tend to distort the meaning of the design views introduced by the methods.

[3]The numbers in brackets correspond to those of the bibliographical references in clause 8.

7. Design methods

IEEE Std 1016-1987 defines the minimal information that shall be captured in a software design description, regardless of the design method used. To illustrate the intent of IEEE Std 1016-1987, this portion of this guide provides examples of relationships for particular design methods to the standard, and illustrates the need to expand methods or integrate them into comprehensive methodologies.

Table 1 in the IEEE Std 1016-1987 presents a table of recommended design views. In this guide tables 1–5 present the same table except that the Example Representation column is replaced by the names of the individual examples.

7.1 Function-oriented design methods

Function-oriented design methods model the software system by breaking it into components, identifying the inputs required by those parts and the outputs produced by them.

Function-oriented design methods include the following:

a) Structured analysis (see [B3] and [B18])
b) Structured design (see [B17])

The two methods can be used in combination as in [B12].

7.1.1 Example

Structured design built upon structured analysis, as described in [B12], is used in this section as a representative function-oriented design method.

Structured design uses data flow diagrams in the analysis phase, to form physical and logical models of the system. Data stores and data flows in the data flow diagrams are recorded in the data dictionary.

Transaction analysis or transform analysis is then applied to derive the structure chart. The structure chart is then factored and refined to obtain highly cohesive modules that are loosely coupled. Process specifications (see [B12]) are also produced to specify complicated modules.

The following are the four major models, or design representatives, produced by this method:

a) *Data flow diagrams.* These can be decomposed into the context diagram, processes, data stores, sources, sinks, and control flows.
b) *Data dictionary.* This can be decomposed into data flow definitions, file definitions, and the definition of data stores used in processes.
c) *Structure charts.* These can be decomposed into afferent, efferent, transform, and coordinate modules and predefined modules.
d) *Process specifications.* These modules pass data that is further defined in the data dictionary.

Table 1—Example design views for structured design

Design View	Scope	Entity Attributes	Structured Design Representations
Decomposition description	Partition of the system into design entities	Identification, type, purpose, function, subordinates	Structure charts
Dependency description	Description of the relationships among entities and system resources	Identification, type, purpose, dependencies, resources	Structure charts, data flow diagrams
Interface description	List of everything a designer, programmer, or tester needs to know to use the design entities that make up the system	Identification, function, interfaces	Data dictionary, data flow context diagram
Detail description	Description of the internal design details of an entity	Identification, processing, data	Data dictionary, process specification

7.2 Data-oriented design methods

Data-oriented methods are design methods in which program structures are derived from the data structures. Tree diagrams are typically used to represent both the data and program structures.

7.2.1 Example

The Jackson Structured Programming (JSP) method as described in [B9] is used to illustrate data-oriented methods.

The first step in applying this method is to define the data structures. Table 2 identifies example design views as represented by Jackson Structured Programming. After the static data relationships have been identified and representations selected, the input data are mapped onto the output data. Where the structures of input data cannot be easily or directly mapped onto the output data, intermediate data items may be identified and data representations selected. In this case, intermediate data streams are identified and a simplified flow diagram is created to document the high-level data flows.

At this point, the program structure is identified and documented. A program structure chart is most commonly used to document the high-level program structure. The program structure chart consists of a tree of functions or operations that can be performed on the data. This tree-based block diagram identifies each functional component according to its place in the program control hierarchy. The relationship of input data to output data identifies all functions to be implemented in the program.

Each function is usually decomposed into input, essential process, and output subfunctions. Coded modules or units are identified with a one-to-one correspondence with the program structure chart.

Finally, as illustrated in Table 2, the detailed operation of each program module, or unit, is defined and documented. Any documentation method for detailed design is suitable.

Table 2—Example design views for Jackson Structured Programming

Design View	Scope	Entity Attributes	Jackson Structured Programming Representations
Decomposition description	Partition of the system into design entities	Identification, type, purpose, function, subordinates	Data and program structure
Dependency description	Description of the relationships among entities and system resources	Identification, type, purpose, dependencies, resources	Program structure charts
Interface description	List of everything a designer, programmer, or tester needs to know to use the design entities that make up the system	Identification, function, interfaces	Data decomposition
Detail description	Description of the internal design details of an entity	Identification, processing, data	Pseudo code

7.3 Real-time control-oriented design methods

Real-time control-oriented design methods consist primarily of extensions to familiar design paradigms, such as described by Gomma (see [B6] and [B7]), Hatley and Pirbhai (see [B8]), and Neilsen and Shumate (see [B11]). The majority of real-time control-oriented design methods are variations on structured design approaches and have augmented data flow notation to handle the complexities of real-time control-oriented systems. The significant element of commonality in real-time control-oriented design methods is the provision for a view that represents concurrence and the heuristics to develop the view. The primary attribute associated with this view is a task or process. Another element typically found in real-time control-oriented design methods is a provision for the modeling of the communication between tasks or processes.

7.3.1 Example

The Ward and Mellor approach (see [B15]) is used as the sample for the real-time control-oriented design approach. Table 3 demonstrates example design views for this approach. This approach covers both requirements analysis and design activities and produces two groups of models:

a) Models of the system essentials
b) Models of the system Implementation

No less than six modeling activities are prescribed to communicate the system implementation. They include the following:

a) *Processor model.* This describes the allocation of essential model activities to processors and their interprocessor interfaces.
b) *Task model.* This describes the allocation of essential model activities to tasks (i.e., the concurrent activities) and their intertask interfaces.
c) *Interface model.* This describes how the concurrent activities communicate.
d) *Process management system services model.* This describes the system utilities necessary to support the application.
e) *Data management system services model.* This describes the system databases.

f) *Code hierarchy model.* This describes the hierarchical organization of code modules for each task.

The notation used for the first five models includes data flow diagrams augmented with symbology for control information, state transition diagrams, entity relationship diagrams, relational tables, data dictionaries, and textual specifications. The code hierarchy model adds the use of structure charts.

The models are generally developed in the order listed, and the heuristics used to develop each model are extensive.

The entities included in the six respective models are the following:

a) Processors
b) Tasks
c) Interfaces
d) Utilities
e) Data
f) Modules and shared data

Table 3—Example design views for Ward and Mellor design

Design View	Scope	Entity Attributes	Ward and Mellor Representations
Decomposition description	Partition of the system into design entities	Identification, type, purpose, function, subordinates	Requirements traceability matrix, data flow diagrams, structure chart
Dependency description	Description of the relationships among entities and system resources	Identification, type, purpose, dependencies, resources	State transition, diagrams, structure chart, data flow diagrams
Interface description	List of everything a designer, programmer, or tester needs to know to use the design entities that make up the system	Identification, function, interfaces	Data flow diagrams, transform specifications, screen layouts, data dictionary, structure chart
Detail description	Description of the internal design details of an entity	Identification, processing, data	Data dictionary

7.4 Object-oriented design methods

An object-oriented design method produces software architectures based on the objects manipulated by systems or subsystems rather than functions. An object-oriented design closely resembles a model of reality since it captures the real-world objects and the operations taken by or upon them. As defined by Booch (see [B1]), an object, is an entity that has the following:

a) A state that denotes its current value
b) A description of the action it suffers or that it requires of other objects
c) A unique instance of some class where a class is characterized by a set of values and a set of operations that are valid for instances of the class
d) A name that can be referenced
e) Controlled visibility to and from other objects
f) The ability to be viewed in two ways
   1) By its outside or behavioral view

2) By its internal or implementation view

An object-oriented design will have a very different topology from one produced with another method, such as a function-oriented method. The resultant design structure will tend to be layers of abstraction where each layer represents a collection of objects with limited visibility to other layers.

7.4.1 Example

A layered virtual machine/object-oriented method (LVM/OOD) is used to illustrate object-oriented methods, as described by Wasserman (see [B16]). The LVM/OOD method ascribes to the concept of parallel decomposition of a system into virtual machines and objects. Each virtual machine provides an instruction set that solves the problem at a given level of understanding. Each instruction in the virtual machine corresponds to either a lower-level virtual machine or an operation on a design object. The design objects provide for definition of data structures and a set of valid operations on the data structure, while the virtual machines describe the processing algorithms. Decomposition of both forms of information in the design proceed in parallel as part of the method.

The LVM/OOD method, as illustrated in table 4, forms the core of the design method. Additional support is provided for designing real-time processing systems. The following steps form the framework for this method:

a) *Determine hardware interfaces.* A context diagram is created that shows which hardware devices must have interface support in the design.
b) *Assign processes to the edge functions.* A separate process is allocated for each hardware device. All other processing is assigned to the middle part.
c) *Decompose the middle part.* Data flow diagrams (DFDs) are used to decompose the middle part into leveled DFDs.
d) *Determine concurrence.* The transforms depicted on the DFDs are combined into processes.
e) *Determine process interfaces.* The form of communication between the processes is determined in this step. The possible communication forms are channels (single or multiple items) and data pools.
f) *Introduce intermediary processes.* The process model is redefined in terms of tasks, where intermediate tasks, such as buffers and transporters, are also introduced.
g) *Encapsulate tasks in packages.* The tasks are analyzed and combined into packages.
h) *Translate design into Program Design Language (PDL).* Details of the processing steps are exposed using PDL.
i) *Decompose large tasks.* Tasks that were allocated to packages in step g) but are too complex to be implemented in a single task body are further decomposed using the LVM/OOD approach.

Table 4—Example design views for LVM/OOD design

Design View	Scope	Entity Attributes	LVM/OOD Representations
Decomposition description	Partition of the system into design entities	Identification, type, purpose, function, subordinates	Data flow diagrams, process structure, charts, finite-state machine, OOD diagrams
Dependency description	Description of the relationships among entities and system resources	Identification, type, purpose, dependencies, resources	Data flow diagrams, process structure charts, finite-state machine, OOD diagrams
Interface description	List of everything a designer, programmer, or tester needs to know to use the design entities that make up the system	Identification, function, interfaces	Context diagram, data flow diagrams, finite-state machines, OOD diagrams
Detail description	Description of the internal design details of an entity	Identification, processing, data	Data flow diagrams, finite-state machines, OOD diagrams

7.5 Formal language-oriented design methods

According to Pedersen (see [B13]): *A minimal criterion that a software development process must meet in order to be called formal is that it must lead to a set of interrelated formal documents. A formal document is one that is written using a formal language, and a formal language is one with a mathematically defined syntax and semantics.*

Some examples of formal language-oriented methods are as follows:

a) Z (see [B14])
b) Paisley (see [B19])
c) Vienna Design Method (VDM) (see [B10])

Thus, to be classed as a formal language-oriented method, a software development process should include a well-defined set of outputs and a formal language in which these outputs are written.

7.5.1 Example

The example formal language-oriented method used here is the Vienna Design Method (VDM). VDM, as illustrated in table 5, is a design approach built around the formal language META IV. It consists of describing the system as a series of models in a layered, top-down style. The key concept driving the models is abstraction, the suppression of irrelevant details. These models, written in META IV, capture at each level all the relevant details of the system at that level.

At the highest level, VDM has a very abstract model, with no representation detail. Yet it describes all the properties needed to express the essential nature of the system. This top-level, abstract model is then transformed into more detailed models that replace the abstractions with more specific details for each subsequent level of transformation.

These models and the associated operations that manipulate them are mathematical objects. The mathematical objects are built, using the type definitions for objects and functions for operations supplied by META IV. Thus the abstraction mechanism used by VDM is based on mathematical concepts rather than the intuition of real-world concepts.

Table 5—Example design views for VDM design

Design View	Scope	Entity Attributes	VDM Representations
Decomposition description	Partition of the system into design entities	Identification, type, purpose, function, subordinates	Semantic function, text description, auxiliary functions
Dependency description	Description of the relationships among entities and system resources	Identification, type, purpose, dependencies, resources	Semantic function, text description
Interface description	List of everything a designer, programmer, or tester needs to know to use the design entities that make up the system	Identification, function, interfaces	Semantic function, text description
Detail description	Description of the internal design details of an entity	Identification, processing, data	Interpretation evolution and elaboration function, semantic domain and state, text description

8. Bibliography

[B1] Booch, G., *Software Engineering with Ada.* Redwood City, CA: Benjamin-Cummings, 1986.

[B2] Chen, P., "The Entity Relationship Model-Toward a Unified View of Data," *ACM TODS*, vol. 1, no. 1, Mar.1976.

[B3] DeMarco, T., *Structured Analysis and System Specification.* Englewood Cliffs, NJ: Yourdon Press, 1987.

[B4] DI-MCCR-80012A, *Software Design Document*, U.S.A., Department of Defense, 1988.

[B5] Gane C. and Sarson, T., *Structured Systems Analysis: Tools and Techniques.* Englewood Cliffs, NJ: Prentice Hall, 1979

[B6] Gomma, H., "A Software Design Method for Real-Time Systems," *CACM,* vol. 27, no. 9, pp. 938–949, Sept. 1984.

[B7] Gomma, H., "Software Development of Real-Time Systems," *CACM*, vol. 29, no. 7, pp. 657–668, July 1986.

[B8] Hatley, D. J., and Pirbhai, I. A., *Strategies for Real Time System Specification.* New York: Dorset House Publication, 1987.

[B9] Jackson, M. A., *Principles of Program Design.* London: Academic Press, 1975.

[B10] Jones, C., *Systematic Software Development Using VDM.* Englewood Cliffs, NJ: Prentice Hall International, 1986.

[B11] Neilsen and Shumate, K., *Designing Large Real Time Systems with Ada.* New York: McGraw-Hill, 1988.

[B12] Page-Jones, M., *The Practical Guide to Structured System Design. (Second ed.)* Englewood Cliffs, NJ: Yourdon Press, 1988.

[B13] Pederson, J. S., "Software Development Using VDM," *SEI Curriculum Module. SEI-CM-161.0*, Software Engineering Institute, Pittsburgh, Apr. 1988.

[B14] Spivey, J.M., "An Introduction to Z and Formal Specifications," *Software Engineering Journal*, Jan. 1989.

[B15] Ward, P. T., and Mellor, S. J., *Structured Development for Real Time Systems. (Vol. 3)* Englewood Cliffs, NJ: Prentice Hall, 1985.

[B16] Wasserman, A., Pircher, P. A., and Muller R. J., "An Object-Oriented Structured Design Method for Code Generation," *Association for Computing Machinery Software Engineering Notes*, vol. 14, no. 1, Jan. 1989.

[B17] Yourdon, E. and Constantine, L., *Structured Design.* Englewood Cliffs, NJ: Prentice Hall, 1979.

[B18] Yourdon, E. N., *Modern Structured Analysis.* Englewood Cliffs, NJ: Yourdon Press, 1988.

[B19] Zave, P., "An insider's view of PAISLEY," *IEEE Transactions on Software Engineering*, vol. 17, no. 3, Mar. 1991.

Annex
Selecting a software design notion

(informative)

(This informative annex is not a part of IEEE Std 1016.1-1993, IEEE Guide to Software Design Descriptions, but is included for information only.)

Design notation is a convention, a system of symbols used to represent concepts, typically in a form of icons. Since selecting design notation to facilitate automation is itself a task, information on it has been provided in this annex. Note that this information is provided for reference only, and not as an attempt to standardize notations or even their choice.

A.1 Data flow diagrams (DFDs)

Data flow diagrams (DFDs) are the origin of structured analysis. A DFD depicts the storage of data within, and the flow of data through, functional requirements of a system. A DFD does not depict its implementation. A DFD is idealized in the sense that processes are assumed to operate instantaneously, inputs are always available when needed, and outputs are always produced. The functional requirements are broken down in a hierarchical manner so that related functions are grouped together and unrelated functions are separated. Each function is nonredundantly specified and partitioned until it can be completely and concisely specified.

A context diagram is the highest level diagram. It shows the boundary between the system and its environment. A context diagram consists of a single data process, which represents the processing to be done by the system, the external entities in the environment with which the system must interact, and the flows representing real-world data through which the system interacts with its environment.

The following objects are provided to support the production of DFDs. Note that either the Yourdon (see [B18]), DeMarco (see [B3]), or Gane and Sarson (see [B5]) symbol set can be used.

a) External entity
b) Data process
c) Data store
d) Split/merge
e) Interface
f) Data flow

In the case of DFDs, these objects may be defined to be either time-discrete or time-continuous and one-way or update flows by virtue of the number of arrowheads added to the end(s) of the flow.

A.2 Transformation graphs (TRGs)

Transformation graphs (TRGs) are diagrams that contain data and/or control information, as proposed by Ward and Mellor (see [B15]) and Hatley and Pirbhai (see [B8]) in their extensions to the Yourdon and DeMarco method to support modeling of real-time systems. Ward and Mellor allow for data and control on the same diagram, whereas Hatley requires data and control on separate diagrams. For a true Hatley method, a pair of diagrams are created with, for example, the same file names suffixed with a *D* and *C* for data and control diagrams respectively. Alternatively, the Hatley control bar can be used instead of the Ward and Mellor control transform object on a single data or control diagram.

Control flow diagrams (CFDs), in the Hatley method, are used to map the flow of control through the system, typically along the same paths as the data flows on the corresponding DFD. Using the Hatley method, the data and control flow diagrams are usually drawn with the same objects at the same positions on each chart, one with data flows and processes and the other with control flows and processes, overlapping each other.

The following objects are provided to support the production of transformation graphs according to the Ward and Mellor or Hatley methods. Note that either the Yourdon, DeMarco, or Gane and Sarson symbol set can be used when creating transformation graphs for either of these methods. In their respective texts, both Ward and Mellor and Hatley and Pirbhai use the Yourdon symbology.

a) External entity
b) Data process
c) Control transform (Ward and Mellor method only)
d) Control bar (Hatley method only)
e) Data store
f) Control store (Ward and Mellor and Hatley methods)
g) Split/merge
h) Interface
i) Data flow
j) Control flow (Ward and Mellor and Hatley methods)

In the case of data and control flows, these may be defined as either time-discrete or time-continuous and one-way or update flows by virtue of the number of arrowheads added to the end(s) of the flow.

A.3 State transition diagrams (STDs)

State transition diagrams (STDs) are diagrams used to show the sequence of states (state machine). A process (control transform or bar) goes through in response to system events (control flow inputs), and the resulting actions (control flow outputs) are what the process performs. This diagram can be used with both the Ward and Mellor and Hatley methods to show the inner functions performed by a control transform or bar. Input control (flow) events are transformed into output control (flow) actions by the process when enabled or triggered.

The following objects are provided to support the production of state transition diagrams according to the Ward and Mellor, and Hatley methods:

a) State
b) Interface
c) Transition

A.4 Structure charts (STCs)

Structure charts (STCs) are diagrams used to portray the structured design of the system in terms of functional hierarchy, dependencies, calling sequences, repetition, inclusion, decisions, and the passing of data and control flow items between the functional entities comprising the system.

The STC should address the following:

a) *Hierarchy.* The depiction that certain functions control other functions.
b) *Modularity.* The collections of instructions that belong together.

c) *Protocol.* The relinquishing of control between parent/child functions.
d) *Information communication.* The passing of data and/or control between processes.
e) *Coupling.* The degree of independence between functions. (This should be minimized.)
f) *Cohesion.* The degree to which activities within functions are interrelated.

The following representation objects are provided to support the production of structure charts according to the Yourdon and Constantine (see [B17]) method:

a) Function
b) Predefined function (library module)
c) Repetitive function (iterative)
d) Decision function
e) Inclusive function
f) Store
g) External device
h) Connection (call)
i) Data flow arrows
j) Control flow arrows
k) Connection continuation
l) Asynchronous call

NOTE—Arrows are sometimes referred to as couples in some texts.

Two types of function and module symbols are provided—one is an outline, and the other is subdivided so that the level number, module name or identifier, etc., may be placed in the upper box and its label placed in the lower box.

Two types of connections are provided—one connection is normal and is used for interfunction calls, and the other is the asynchronous call used to depict, for example, functions invoked in response to system interrupts, etc.

A.5 Entity relationship diagrams (ERDs)

Entity relationship diagrams (ERDs) are used to model the structure and relationships of data typically stored in data stores on data flow diagrams or transformation graphs. The ERD is a conceptual representation of real-world objects and the relationships between them, and defines information the system must process as well as the inherent relationships that must be supported by the database (store). The symbols provided are in accordance with the symbology defined by Chen (see [B2]) and subsequently used by Ward and Mellor and Yourdon in their texts.

The following object types are provided to support the production of ERDs:

a) Entity
b) Relationship
c) Weak entity
d) Attribute
e) Connection
f) Associative relationship
g) Subtype/supertype
h) One-to-one
i) One-to-many/many-to-one

A.6 Objects

The term *object*, as used here, provides graphical expressions and representations of notations as a means of manipulation provided by software CASE tools.

Objects are entities that are placed on the chart and subsequently manipulated. This builds interactions that collectively provide a visual operation of the function and provides interrelationships and structure to the system being defined. An object may be a symbol, connection, flow, transition, arrow, label, or text block. Each object is defined as follows and is appropriate to the type of chart being drawn.

Arrow (or couple) (STC). This is used to show the flow of data or control between symbols and is depicted by an arrow symbol with a circle at the base placed adjacent to a connection between a pair of processes. An unfilled circle is used to depict a data flow, whereas a filled circle depicts a control flow. The arrow points in the direction of flow between process symbols. The arrow label is placed adjacent to the arrow and is typically the same as the corresponding data item connecting the equivalent data process pair on the corresponding data flow diagram.

Associative entity (ERD). This is depicted by a connection with an arrowhead at the end coinciding with a relationship symbol. It represents a relationship that participates in other relationships. Its key consists of those of the participating entities.

Attribute (ERD). This is depicted by a circle and is used to represent data elements that help to describe a particular entity or relationship. This ought to describe only one thing. Attributes that help to uniquely identify an entity or relationship are called key attributes.

Connection (STC). This is depicted by a directed, solid, continuous line connecting a pair of function or module symbols and is terminated with an arrowhead at the destination symbol end. It is used to show a calling sequence or dependency between symbols on a structure chart. A data connection is a solid, continuous line between a module or function symbol; is a store or external device on an STC; and is unterminated. A connection is generally unlabeled though labels could be used to denote options in the case of a decision process invoking alternative subprocesses. An asynchronous call is depicted using a broken line and may be used for invoking function, etc., in response to a system interrupt.

Continuation (STC). This is placed at the end of a connection, and is used to denote that the connection continues elsewhere on the same chart. It is used to prevent unnecessary crossing of connections. This symbol is a small rectangle.

Control bar (Hatley TRG). This is depicted by a vertical bar symbol with no name. There may be more than one bar on a diagram, but it represents only one control specification.

Control flow (TRG only). This is depicted by a directed, broken, continuous line between a pair of symbols or an interface on a TRG and is terminated by one or two arrowheads at one or both ends. It is used to denote the transfer of an event, trigger or enable/disable between symbols and the data flow may be time-discrete, time-continuous, and one-way or update flow. The flow name is placed on top of, or adjacent to, the flow and follows the same naming rules as for data flows.
Rules: Control flows may only flow between two data and/or control processes, a terminator and a data or control process, a process and a control bar (Hatley method) or a store and a control bar (Hatley method).

Control store symbol (Ward and Mellor TRG). This is depicted by a pair of horizontal, broken lines and is used to store a control flow output by a source data process or control

transform (or bar) until a destination data process or control transform (or bar) requires it. The store name and identifier is placed between the lines. A control store is typically further defined by a control table.

Control transform symbol (TRG). This is depicted by a circle with a broken (dashed) outline and is used to transform input control flows into output control flows only. The transform name (verb-noun), level number, and/or identifier is placed inside the symbol. The identifier is typically a number that identifies the hierarchical standing of the control transform in the system (for example, 1.1.2). Control transforms are typically further modeled, using a state transition diagram, structured decision table, or process activation table.

Data flow (DFD or TRG). This is depicted by a directed, solid, continuous line between a pair of symbols or an interface on a DFD or TRG and is terminated at one or both ends by one or two arrowheads. It is used to denote the transfer of a data item (a "pipeline") between symbols and may be time-discrete (single arrowhead at destination symbol) or time-continuous (double arrowhead at destination symbol), and may be either a one-way (arrowhead(s) at destination symbol only) or an update (arrowhead(s) at source and destination symbols) flow. The flow name is placed on top of, or adjacent to, the flow and must be unique, including only nouns and adjectives to represent its contents. No processing should be implied.
Rules: Data flows may only flow between two processes, a terminator and a process or a store and a process, in either or both (update) directions.

Data flow diagram. A symbol may be an external entity (terminator), data process, data store, split/merge, or interface. Data flows are used to connect symbols.

Data process symbol (DFD or TRG). This is depicted by a circle with a solid outline and transforms input data (and control flows) into output data (and control flow). The process name (verb-noun), level number, and/or identifier and/or placed inside the symbol. The identifier is typically a number that identifies the hierarchical standing of the data process in the system (for example, 1.2.3). Data processes are typically further modeled by either a lower-level DFD or TRG or a primitive process specification (PPS) that may be in the form of structured English, pseudo code (or PDL) or a flowchart, etc.

Data store symbol (DFD or TRG). This is depicted by a pair of horizontal, solid lines and is used to store a data flow output by a source data process until a destination data process requires it. The store name (unique) and identifier is placed between the lines and should be a noun and descriptive adjective(s); no processing should be implied. A data store should ideally appear on only one diagram (though it may appear in more than one place on that diagram), preferably at the highest level at which it is used. Data stores are typically further modeled, if sufficiently complex and/or structured, using an entity-relationship diagram.

Direction indicator (ERD). This is depicted by a connection with an arrowhead and represents the direction of access of data in a database.

Entity (ERD). This is depicted by a rectangle, with a unique singular noun name, and represents an object or group of objects about which information is to be stored or collected. An entity has a significant purpose or characteristic within the system (data view) being modeled.

Entity relationship diagram. A symbol may be an entity, relationship, weak entity or associative entity. Symbols are connected and may be one-to-one, one-many, many-to-one, many-to-many, or subtype/supertype.

External entity (or terminator) symbol (DFD or TRG). This is depicted by a rectangle and is used to portray a physical device or other system that is outside the scope of the system being defined, and provides information used by the system or uses information produced by the system. The name of the device is placed inside the symbol.

Function (STC). This is used to represent a collection of statements or operations that will implement a particular function. The symbol used is a rectangle inside of which the process name is placed. An alternative symbol is a rectangle with a horizontal line inside dividing it into two parts—the upper, which can be used to contain a level number and/or identifier and the lower, which contains the process name.

Interface symbol (DFD, TRG, or STD). This is depicted by a very small rectangle and is used to show that the attached connection either originates from, or is being sent to, a symbol outside the scope of the current chart. It is connected to a symbol by a data or control flow on a data flow.

Module (STC). This is a process that is outside the system and has already been developed in the form of a library function or operating system function, etc. It is similar to the function symbols, but the left and right sides each consist of closely spaced vertical lines.

Relationship (ERD). This is depicted by a diamond, with a verb name, and represents a significant association or interaction involving one or more entities. A relationship cannot stand alone and must be connected to at least two entities with a line (ERD connection). Two aspects of this connection need to be accounted for, namely cardinality, the rate of relative occurrence of one entity in the relationship to another (one or many), and exclusivity, whether an entity must have a relationship with one or more entities in the relationship (some or all).

Split/merge symbol (DFD or TRG). This is depicted by a small circle and is used to split a composite flow into its component flows, to merge component flows into a composite flow, or to enable a single flow to be directed to several separate processes.

State symbol (STD). This is depicted by a rectangle and is used to show a stable system state, during which processing continues until the occurrence of one or more specified transition events. A system may only be in one state at a time and must always be in a state.
Rules: Each state should be reachable from the start state. The end state should be reachable from any state. From a given state, a specific event can cause a transition to only one other state.

State transition diagram. A symbol may be a state or interface. A transition is used to connect symbols.

Structure chart. A symbol may be a function, predefined function, repeat function, decision function, inclusive function, module, predefined module, store, device, asynchronous call, or continuation. In addition, an arrow is also an object for this chart type. A connection is used to connect symbols.

Subtype/supertype (ERD). This is depicted by connection with a perpendicular bar across the middle segment. It is a means of identifying subcategories of an entity. The subentity inherits all the properties of the superentity and has, in addition, unique properties of its own.

Text block. An object that can be used on any chart type, generally for annotation or heading purposes.

Transformation graph. A symbol may be an external entity, data process, control transform, data store, control store, split/merge, interface, or control bar. A data flow or control flow is used to connect symbols.

Transition (STD). This is depicted by a directed, solid, continuous line connecting a pair of state symbols, terminated with an arrowhead at the destination symbol end. It is used to show a change of system state on a state transition diagram. The condition or event that caused, and action that results from, the change of state are placed adjacent to the transition and prefixed with C: and E:, respectively.