IEEE P1016/D5.0; 12 December 2005

IEEE Draft Standard for Software Design Descriptions

Sponsor

Software & Systems Engineering Standards Committee
of the

IEEE Computer Society

Abstract: The required information content and organization for Software Design Descriptions (SDDs) are described. An SDD is a representation of a software design to be used for communicating design information to its stakeholders. The requirements for the design languages (notations and other representational schemes) to be used for conformant SDDs are specified. This standard is applicable to automated databases and design description languages but can be used for paper documents and other means of descriptions.
Keywords: design state, design subject, design view, design viewpoint, software design, software design description, diagram.

IEEE Standards Activities Department
Standards Licensing and Contracts
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA

Introduction

(This introduction is not part of IEEE P1016, IEEE Standard for Software Design Descriptions.)

This standard specifies requirements on the information content and organization for Software Design Descriptions (SDDs). An SDD is a representation of a software design that is to be used for recording design information addressing various design concerns and communicating that information to the design's stakeholders.

SDDs play a pivotal role in the development and maintenance of software systems. During its lifetime, an SDD is used by acquirers, project managers, quality assurance staff, configuration managers, software designers, programmers, testers, and maintainers. Each of these stakeholders has unique needs, both in terms of required design information and optimal organization of that information. Hence, a design description will contain the design information needed by those stakeholders.

The standard also specifies requirements on the design languages to be used when producing SDDs conforming to these requirements on content and organization.

The standard specifies that an SDD be organized into a number of design views. Each view addresses a specific set of design concerns of the stakeholders. Each design view is prescribed by a design viewpoint. A viewpoint identifies the design concerns to be focused upon within its view and selects the design languages used to record that design view. The standard establishes a common set of viewpoints for design views, as a starting point for the preparation of a SDD, and a generic capability for defining new design viewpoints thereby expanding the expressiveness of an SDD for its stakeholders.

This standard is intended for use in design situations in which an explicit software design description is to be prepared. These situations include traditional software design and construction activities leading to an implementation as well as "reverse engineering" situations where a design description is to be recovered from an existing implementation.

This standard can be applied to commercial, scientific, military and other types of software. Applicability is not restricted by size, complexity, or criticality of the software. This standard considers both the software and its system context, including the developmental and operational environment. It can be used where software comprises the system or where software is part of a larger system characterized by hardware, software and human components and their interfaces.

This standard is applicable whether the SDD is captured using paper documents, automated databases, software development tools or other media. This standard does not explicitly support, nor is it limited to, use with any particular software design methodology or particular design languages, although it establishes minimum requirements on the selection of those design languages.

This standard is consistent for use with IEEE/EIA Std 12207.0–1996, Software Life Cycle; it may also be applied in other life cycle contexts.

This standard consists of six clauses.

Clause 1 defines the scope and purpose of the standard.

Clause 2 references documents containing material required to understand and apply the standard.

Clause 3 provides definitions of terms used within the context of the standard.

Clause 4 provides a framework for understanding software design descriptions in the context of their preparation and use.

Clause 5 describes the required content and organization of an SDD.

Clause 6 defines several design viewpoints.

Annex A provides a bibliography.

Annex B defines how a design language to be used in an SDD may be described in a uniform manner.

Annex C contains templates for organizing an SDD conforming to the requirements of this standard.

This standard follows the *IEEE Standards Style Manual*. In particular, the word *shall* identifies requirements that must be satisfied in order to claim conformance with this standard. The verb *should* identifies recommendations and the verb *may* is used to denote that particular courses of action are permissible.

This standard is modeled after IEEE Std 1471, extending it primarily to support detailed design and construction for software. The demarcation between architecture, high-level and detailed design is arbitrary for small to medium sized systems. While IEEE 1471 recommends the use of certain viewpoints, P1016 requires the use of specific viewpoints.

At the time this standard was completed, the working group had the following membership:
Vladan V. Jovanovic, *Chair (acting)*
Basil A. Sherlund, *Chair (emeritus)*
Rich Hilliard, *Secretary and Technical Editor*

Nenad Anicic	Philippe Kruchten	Stevan Mrdalj
Edward Byrne	Kathy Land	Ira Sachs
Bob Cook	Joaquin Miller	Judith Speights
Ed Corlett	James Moore	

The following members of the balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention. (To be provided by IEEE editor at time of publication.)

TABLE OF CONTENTS

IEEE Draft Standard for Software Design Descriptions

1. Overview

1.1 Scope

This is a standard for *software design descriptions* (SDD). An SDD is a representation of a software design to be used for recording design information and communicating that design information to key design stakeholders.

This standard is intended for use in design situations in which an explicit software design description is to be prepared. These situations include traditional software construction activities, when design leads to code, and "reverse engineering" situations where a design description is to be recovered from an existing implementation.

The standard can be applied to commercial, scientific, or military software that runs on digital computers. Applicability is not restricted by the size, complexity, or criticality of the software. This standard can be applied to the description of high-level and detailed designs.

This standard is not limited to use with specific methodologies for design, configuration management, or quality assurance. This standard does not require the use of any particular design languages, but establishes requirements on the selection of design languages for use in an SDD. The standard can be applied to the preparation of SDDs captured as paper documents, automated databases, software development tools or other media.

NOTE—The requirements in P1016 are intended to be consistent with the use of other IEEE standards (specifically, IEEE 830, IEEE 1012, IEEE 1471 and IEEE/EIA 12207).

1.2 Purpose

This standard specifies requirements on the information content and metadata organization of SDDs. This standard specifies requirements for the selection of design languages to be used for software design description, and requirements for documenting design viewpoints to be used in organizing a software design description.

1.3 Intended Audience

This standard is intended for technical and managerial stakeholders who prepare and use SDDs. It will guide a designer in the selection, organization, and presentation of design information. For an organization developing its own design description practices, the use of this standard will help to ensure that design descriptions are complete, concise, consistent, interchangeable, appropriate for recording design experiences and lessons learned, well organized and easy to communicate.

1.4 Conformance

A Software Design Description conforms to this standard if it satisfies all of the requirements of this standard. Requirements are denoted by the verb *shall*.

2. References

None

3. Definitions

For the purposes of this standard, the following terms and definitions apply. *The IEEE Standard Dictionary of Electrical and Electronics Terms* [IEEE Std 100], and *Industry Implementation of International Standard ISO/IEC 12207:1995 Software life cycle processes* [IEEE/EIA Std 12207.0–1996] should be referenced for terms not defined in this clause.

design concern: an area of interest with respect to a software design.

design constraint: an element of a design view which names and specifies a rule or restriction on a design entity, design attribute or design relationship. **See**: design entity, design relationship, design attribute

NOTE—May need to reconcile with an earlier IEEE definition: design constraint: Any requirement that affects or constrains the design of a software system or software system component (for example, physical requirements, performance requirements, software development standards, software quality assurance standards). [ANSI/IEEE Std 610.12-1990]

design element: an item occurring in a design view which may be any of the following: design entity, design relationship, design attribute, or design constraint.

design attribute: an element of a design view which names a characteristic or property of a design entity, design relationship or design constraint. **See**: design entity, design relationship, design constraint

design entity: an element of a design view which is structurally, functionally or otherwise distinct from other elements, or plays a different role relative to other design entities. **See**: design view

design overlay: a representation of additional, detailed or derived design information organized with reference to a previously-defined design view.

design rationale: information capturing the reasoning of the designer which led to the system as designed, including design options, tradeoffs considered, decisions made, and the justifications of those decisions.

design relationship: an element of a design view which names a connection or correspondence between design entities. **See**: design entity

design stakeholder: an individual, organization or group (or classes thereof playing the same role) having an interest in, or design concerns relative to, the design of some software item. **See**: design concern

design subject: any software item or system which is to be constructed or which already exists and is to be analyzed, for which a software design description will be prepared. **Alternate terms to consider**: *software under design* or *system under design*.

designer: the stakeholder responsible for devising and documenting the software design.

design view: a representation comprised of one or more design elements to address a set of design concerns from a specified design viewpoint. **See**: design concern, design element, design viewpoint

design viewpoint: a specification of the elements and conventions available for constructing and using a design view. **See**: design view

diagram (type): a logically coherent fragment of a design view, using selected graphical icons and conventions for visual representation from an associated design language, to be used for representing selected design elements of interest for a system under design from a single viewpoint **See:** design subject

4. Conceptual Framework for Software Design Descriptions

This clause establishes a conceptual framework for Software Design Descriptions. The conceptual framework includes basic terms and concepts of software design description, the context in which SDDs are prepared and used, the stakeholders who use them, and how they are used.

4.1 Software Design in Context

A *design* is a framework which demonstrates a means to fulfill the requirements for some software item and to guide the implementation of that software item. A *design subject* is any software item to be constructed or which already exists and is to be analyzed, without loss of generality we will also refer to a design subject as the *system under design* or *software under design*. This standard does not establish what a design subject may be. Examples of design subjects include systems, subsystems, applications, components, libraries, application frameworks, application program interfaces (APIs) and design pattern catalogs.

A *software design description* (SDD) is a representation of some design subject of interest. An SDD is prepared to represent exactly one design subject. An SDD can be produced to capture one or more levels or layers of concern with respect to its design subject. These levels or layers are usually defined by the design methods in use or the life cycle context; they have names such as architectural design, logical design, or physical design. An SDD can be prepared and used in a variety of design situations. Typically, an SDD is prepared to support the development of a software item to solve a problem, where this problem has been expressed in terms of a set of requirements. The contents of the SDD can then be traced to these requirements. In other cases, an SDD can be prepared to understand an existing system lacking any design documentation. Typically, there is a system under development or under review for which an SDD is to be described such that information of interest is to be captured, organized, presented and disseminated to all interested parties. This information of interest can be used for planning, analysis, implementation and evolution of the software system, by identifying and addressing essential design concerns. A *design concern* is any area of interest in the design, pertaining to its development, implementation, or operation. Design concerns are expressed by *design stakeholders*—those parties which may be individuals, groups and organizations with an interest in the design of the system. Frequently design concerns arise from specific requirements on the software, others arise from contextual constraints. Typical design concerns include functionality, reliability, performance, and maintainability. Typical design stakeholders include users, developers, software designers, system integrators, maintainers, acquirers, and project managers.

NOTE—From a system-theoretic standpoint, an SDD is the information content of the design state space with convenient inputs (typically design diagrams and specifications produced by designers) and outputs (results of transformations typically produced by software tools). System state in case of design information typically contains alternative designs and design rationale information in addition to the minimal information of the current version of design. An interesting property of a design description as a system is that its configuration is subject to dynamic evolution and the respective state space (based on its design elements) is not given in advance but created iteratively in a manner of system analysis by synthesis. The final design (synthesis) is obtained via successive analysis of intermediate designs. Therefore, a design description can be considered an open, goal-directed system whose end state is a detailed model of the system under design.

An SDD is organized using *design views*. A design view addresses one or more of the design concerns.

NOTE—The use of multiple views to achieve separation of concerns has a long history in software engineering: [Ross-Goodenough-Irvine, 1975], recently in *viewpoint-oriented* requirements engineering [Finkelstein, 1992], and particularly relevant to this standard, the use of views to rationally present the results of a design process [Parnas, Clements 1986], and their use during the design [Gomma, O'Hara 1998]. The particular formulation here is built upon that found in IEEE Std 1471.

Each design view is governed by a *design viewpoint*. Each design viewpoint focuses on a set of the design concerns and introduces a set of descriptive resources (or *view elements*) that are used to construct and interpret the design view. E.g., a viewpoint may introduce familiar elements such as functions, input and outputs; these elements are used to construct a functional view.

There are four kinds of view elements: *design entities*, *design relationships* among entities, *design attributes* and *design constraints* on those elements. A design viewpoint will define the view element types to be used in any design views it governs. Each design view used to represent a software system is expressed as a collection of instances of design entities, attributes, the relationships among design entities and constraints on those elements. The design information needs of stakeholders of the system under design are to be satisfied through use of these elements.

NOTE—Although a view is not a graph, its content is frequently described using diagrams which may be formalized as extensions or specializations of conceptual graphs of design elements. [Draft ISO Std Conceptual Graphs 2001].

It is sometimes useful to to gather and present information which does not strictly following the partition of information by viewpoints. A *design overlay* is a mechanism intended to organize and present design additional, detailed or derived information with respect to an already-defined design viewpoint for this purpose.

It is not sufficient to document only the actual design; it is also useful to capture the *design rationale* including alternative designs and the justifications for choices which have been made, whenever experience suggests long-term relevance and value of such information for current and future stakeholders.

The key concepts of software design description are depicted in figure 1.

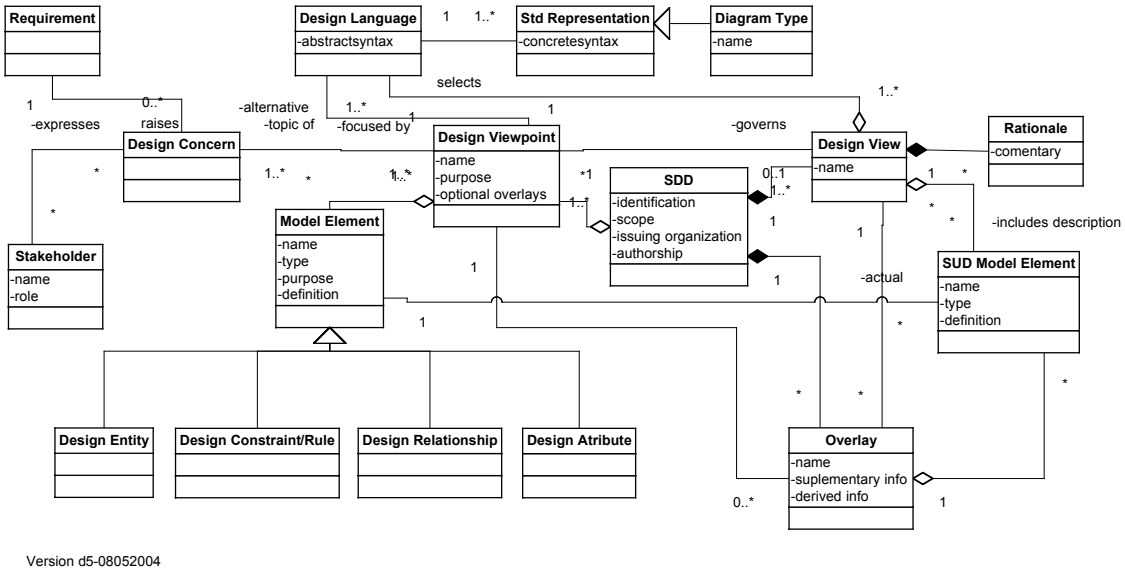


Figure 1. Conceptual Framework of Software Design Description

NOTE—Figure 1 provides a summary of the key concepts used in this standard and their relationships. The figure presents these concepts and their relationships in the context of a single SDD depicting a single design subject. An SDD comprises a set of Design Views from different Viewpoints selected to cover all (stakeholder) concerns as (design) Requirements. Specific content of each View is expressed in terms of Entities, their Attributes and involved Relationships, using a selected Design Language. In the figure, boxes represent classes of things. Lines connecting boxes represent associations between classes of things. Each class participating in an association has a role in that association. A role can optionally be named with a label, appearing close to where the association is attached to the class. For example, in the association between Software Design Description and Design Subject, the role of Software Design Description is labeled describes. Each role may have a multiplicity, denoting a number or set of numbers such as a numeric range. A diamond (at the end of an association line) denotes a part-of relationship. For example, the association between Design View and Software Design Description should be interpreted to read "one or more Design Views are part of a Software Design Description". This notation is defined in the Unified Modeling Language Specification [UML 2.0].

To facilitate automation, exchange and long-term relevance of SDDs, the design state and design rationale information is accompanied by metadata describing both design state and design rationale. Metadata are organized around viewpoints and design state around design views to include instances for design entities, attributes and relationships. Figure 2 depicts the state transitions of an SDD in this respect.

Most importantly, this standard assumes use of models in software design. Models and their representations can be used in different modes: as sketches or rough drafts; as blueprints suitable for implementation; and as executable specifications. The use of models as sketches, while highly recommended in practice, is not governed by this standard; the intended modes are blueprints and executable specifications, as formal engineering documents. The primary use of a sketch is as an aid for thinking, and in conversation about ideas before those ideas can be systematized into designs as either blueprints or executable specifications. Blueprints are developed under general requirements of consistency and reasonable completeness and intended to communicate designs to humans such as to implementers or to maintainers trying to understand the design in order to change it. Executable specifications further restrict descriptions to those that can be automatically translated into implementations on real machines, without the intervention of human intelligence. Software design descriptions covered by this standard are not only formalized using defined languages, but are also intended to be precise i.e., rigorous irrespective of the medium to be used to record them. It is the intent to communicate specific ideas only and not to present complete designs to be implemented as is, that distinguishes sketches from blueprints and executable specifications.

There is no restriction by this standard to the use of any design language in sketches or to the use of sketches in documentation, but the expectation is to use the P1016 standard to govern blueprints and/or executable specifications in the full scope of design responsibility. In the anticipated lifetime of this standard (2005 to 2010), design automation is far more feasible and likely than in paper-based designs, but that expectation by no means eliminates the convenience of sketches (including paper, whiteboards and PC-tablets to capture them) as designers are humans and the purpose of design information is human communication, particularly for the purpose of critiquing designs.

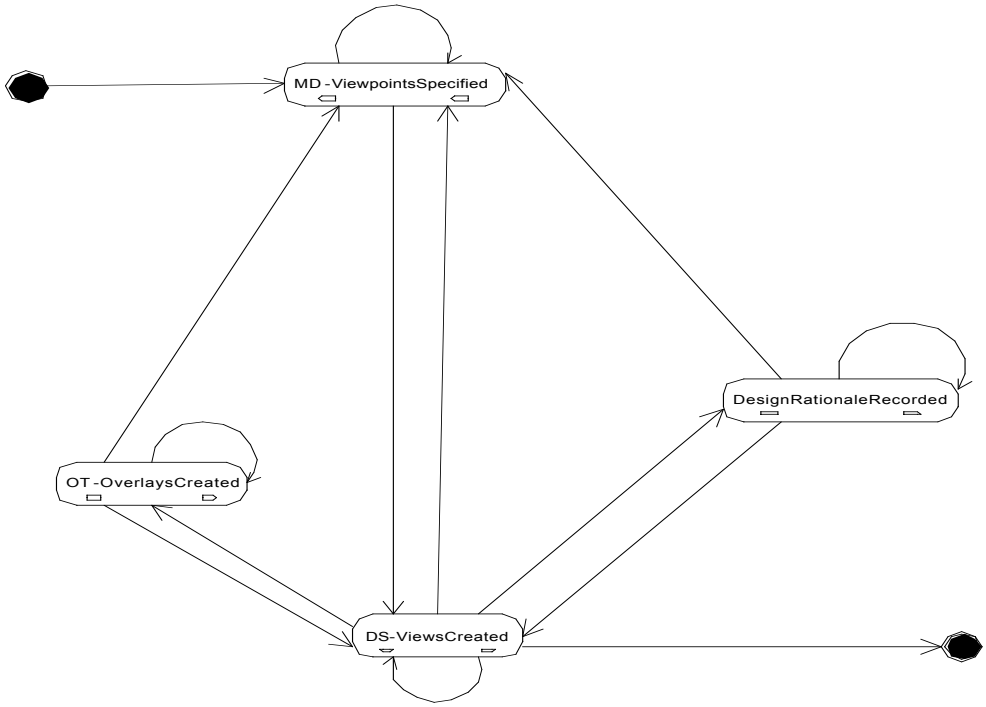


Figure 2. State Transitions of an SDD

4.2 Software Design Descriptions within the Life Cycle

In this standard, a typical cycle will be used to describe the various design situations in which an SDD can be created and used. This life cycle is based on IEEE/EIA 12207.

4.2.1 Influences on SDD Preparation

The key software life cycle product that drives a software design is typically the software requirements description (SRD). An SRD captures the software requirements which will drive the design, and may contain design constraints that must be considered or observed.

4.2.2 SDD Influences on Software Life Cycle Products

The SDD influences the content of several major software life cycle products. Developers of these products will be recognized among the SDD's intended audience.

- Software Requirements Description. Design decisions, or design constraints discovered during the preparation of the SDD, may lead to requirements changes. Often traceability between requirements and design is maintained to manage these changes.
- Test Documentation. Test planning can be influenced by the SDD, but any white-box testing activities at the level of unit, integration, and system testing, are directly influenced by the SDD. Developers of any test specifications and test cases that relate to this type of testing should cover the design functionality, relationships, objects, and data descriptions contained in the SDD.

4.2.3 Design Verification and Design Role in Validation

Verification is a process for determining whether the software products of an activity fulfill the requirements or conditions imposed on them in the previous activities. [IEEE/EIA 12207.0] An SDD can be subject to design verification to ascertain whether the design: is consistent with stated requirements; implements intended design decisions (such as those pertaining to interfaces, inputs, outputs, algorithms, resource allocation, and error handling); achieves intended qualities (such as safety, security, maintainability); and conforms to an imposed architecture. Verification therefore raises a set of design concerns which can be dealt with in the SDD and subjected to inspection or analysis.

Validation is a process for determining whether the requirements and the final, as-built system or software product fulfills its specific intended use. [IEEE/EIA 12207.0] The SDD can play a role in this process mainly by providing: an overview necessary for understanding the implementation; the rationale justifying design decisions made; and traceability back to the requirements on the software item.

5. Design description information content

5.1 Introduction

The required elements of an SDD are:

- an identification of the SDD,
- its identified stakeholders,
- its identified design concerns,
- its selected design viewpoints, each with type definitions of its allowed design elements and design languages,
- its design views,
- its design overlays, and
- its design rationale.

These are described in the remainder of this clause.

5.2 SDD Identification

An SDD shall include the following descriptive information:

- date of issue and status;
- scope;
- issuing organization;
- authorship (responsibility or copyright information);
- references;
- context;
- one or more design languages for each design viewpoint used;

- body;
- summary;
- glossary;
- change history.

NOTE—This requirement enables an SDD to conform with the IEEE/EIA Std 12207.1 Item Content Guidelines. The description of *design languages* will be a proper part of the design viewpoint declarations (5.5). The *body* of the SDD will be organized into design views (5.4), possibly with associated overlays (5.7) and design rationale (5.8).

5.3 Design Stakeholders and Concerns

An SDD shall identify the stakeholders for the design subject.

An SDD shall identify the design concerns for each stakeholder.

An SDD shall address each identified design concern. In addition, an SDD shall address the following design concerns when applicable to the system under design:

- purpose: describe the design of a software item. (the software design description and the architecture description provides the detailed design needed to implement the software.) may be supplemented by software item interface design and database design.
- description of how the software item satisfies the software requirements, including algorithms and data structures;
- software item input/output description;
- static relationships of software units;
- concept of execution, including data flow and control flow;
- requirements traceability: 1) software component-level requirements traceability; 2) software unit-level requirements traceability;
- rationale for software item design;
- reuse element identification.

NOTE—This requirement enables an SDD to meet the software design description requirements of [IEEE/EIA 12207.1 (6.16)]:

5.4 Design views

A software design description shall be organized into one or more design views. A *design view* is a representation consisting of design entities, design entity attributes, design relationships and design constraints to address an identified set of design concerns from a specific viewpoint.

The purpose of a design view is to address design concerns pertaining to the design subject, to allow a design stakeholder to focus on design details from a different perspective or design viewpoint, and effectively address relevant requirements.

Design views are the means of organizing an SDD to satisfy the needs of each design stakeholder and to promote separation of concerns. Each design view addresses one or more design concerns. Together, these views provide a comprehensive description of the design in a concise and usable form that simplifies information access and assimilation. Each software design stakeholder can have a distinct perspective on what are the essential aspects of a software design. Other design information may be extraneous to that stakeholder.

An SDD is *complete* when each identified design concern is the topic of at least one design view, all design attributes refined from each design concern by some viewpoint have been specified for all of the design entities and relationships in its associated view and all design constraints have been applied.

An SDD is *consistent* if there are no known conflicts between the elements of its design views.

NOTE—Users of P1016 may wish to state delivery requirements on an SDD in terms of the notions of completeness and consistency.

5.5 Design viewpoints

A design viewpoint is a specification of the conventions for constructing and using a design view. It identifies the resources from which to develop individual design views. For each design view in an SDD, there shall be a design viewpoint governing it.

Each design viewpoint shall be specified by:
- the viewpoint name;
- the concerns which are the topics of the viewpoint;
- the resources, or view elements, provided by that viewpoint, specifically the types of design entities, attributes, relationships and constraints introduced by that viewpoint or used by that viewpoint (which may have been defined elsewhere). These elements may be realized by one or more design languages;
- analytical methods or other operations to be used in constructing the view based upon the viewpoint, and criteria for evaluating the design based upon the viewpoint; and
- the viewpoint source (e.g., authorship or citation) when applicable.

In addition, a design viewpoint specification may provide the following information on using the viewpoint:
- formal or informal consistency and completeness tests to be applied to the models making up an associated view;
- evaluation or analysis techniques to be applied to the models; and
- heuristics, patterns, or other guidelines to assist in construction or synthesis of an associated view.

An SDD shall include a rationale for the selection of each selected viewpoint.

Each design concern identified in an SDD shall be addressed by at least one viewpoint selected for use. A design concern may be the focus of more than one viewpoint in an SDD.

NOTE—A design viewpoint specification may be included in the SDD or incorporated by reference.

NOTE—It is envisioned that through the selection of suitable viewpoints an SDD can achieve conformance to other development standards.

5.6 Design elements

A design element (or model element) is any item occurring in a design view. A design element may be any of the following: design entity, design relationship, design attribute, or design constraint.

Each design element shall have the following attributes: a name, a type and an expression.

The type of each design element shall be introduced within exactly one design viewpoint definition.

A design element may be referenced in one or more design views.

NOTE—A design element is introduced and "owned" by exactly one design view; conforming to its type definition within the associated viewpoint definition. It may be shared or referenced within other design views. Sharing of design elements permits the expression of design aspects. [Aspect-Oriented Programming]

5.6.1 Design entities

Design entities capture key elements of a software design. Each design entity shall bear a unique name and may be referenced by that name throughout the SDD. The intent of design entities is to divide the design

subject into separate elements that can be considered, implemented, changed, and verified with minimal effect on other entities.

Entities can represent systems, subsystems, libraries, frameworks, abstract collaboration patterns, generic templates, components, classes, data stores, modules, programs, and processes.

The number and type of entities needed to express a design view are dependent on a number of factors, such as the complexity of the system, the design technique used, and the programming environment.

Although entities are different in nature, they possess common characteristics. Each design entity will have a name, type, and purpose. The common characteristics of entities are described by design entity attributes (5.7). There are common relationships among entities such as interfaces or shared data (5.8).

5.6.2 Design attributes

A *design attribute* is a named characteristic or property of a design entity, design constraint, or a design relationship. It provides a statement of fact about the design element. Design attributes can be thought of as questions about design elements. The answers to those questions are the values of the attributes. All the questions can be answered, but the content of the answer will depend upon the nature of the entity. The collection of answers provides a complete description of an entity.

All attributes declared by a design viewpoint shall be specified. Attribute descriptions should include references and design considerations such as tradeoffs and assumptions when appropriate. In some cases, attribute descriptions may have the value *none*.

- The design attributes defined in 5.7.1 through 5.7.3 shall be applied to all design entities used in an SDD.

NOTE—Design attributes have been generalized from the concept of *design entity attribute* (which appeared in IEEE 1016-1998 and applied only to design entities) to apply to design entities, design relationships and design constraints.

NOTE—The design attributes listed below insure compatibility with IEEE 1016-1998. Other design attributes required as a part of specific design viewpoints are defined with those viewpoints (6). Some design attributes (such as subordinates [see 6.2.2.1] can be more usefully represented as design relationships. This was not possible in IEEE 1016–1998.

5.6.2.1 Unique naming attribute

The name of the element. All design elements shall have a name. Each element shall have an unambiguous reference name. The names for the elements may be selected to characterize their nature. This will simplify referencing and tracking in addition to providing identification.

5.6.2.2 Entity type attribute

A description of the kind of element. The type attribute shall describe the nature of the element. It may simply name the kind of element, such as subsystem, component, framework, library, class, subprogram, module, function, procedure, process, object, persistent object, class, or data store. Alternatively, design elements may be grouped into major classes to assist in locating an element dealing with a particular type of information. For a given design description, the chosen element types shall be applied consistently.

5.6.2.3 Purpose attribute

A description of why the element exists. The purpose attribute shall provide the rationale for the creation of the element.

5.6.2.4 Author attribute

Identification of designer. The author attribute shall identify the author of the element, as an individual, or the organization responsible for design description.

5.6.3 Design relationships

A *design relationship* is a named association or correspondence among two or more design entities. It provides a statement of fact about those design entities.

NOTE—There are no required design relationships in this standard, however most design techniques use design relationships extensively. Normally these design relationships will be defined as a part of a design viewpoint. For example, structured design methods are built around design relationships including input (datum *I* is an **input** to process *A*), output (datum *O* is an **output** of process *A*) and decompose (process *A* **decomposes** into processes *A1*, *A2* and *A3*) relationships. Object-oriented design methods use design relationships including encapsulation, generalization, specialization, composition, aggregation, realization and instantiation.

5.6.4 Design constraints

A *design constraint* is an element of a design view which names a rule or restriction imposed on another design element which may be a design entity, design attribute or design relationship.

5.7 Design overlays

A *design overlay* is a mechanism for presenting additional, detailed or derived information with respect to an already-defined design view. It is frequently convenient to capture such information, as an alternative to introducing a new viewpoint, using overlays upon a subset of the information in the diagrams selected in existing relevant viewpoints.

Each design overlay shall be clearly marked.

Each design overlay shall be clearly associated with a single viewpoint.

NOTE—Reasons to utilize a design overlay as a part of an SDD include: to provide an extension mechanism for design information to be presented conveniently on top of some view without a requirement for existing external standardization of languages and notations for such representation; to extend expressive power of representation with additional details while reusing information from existing views (i.e. without a need to define additional views or persistently store derivable design information); and to relate design information with facts from the system environment for the convenience of the designer (or other stakeholders).

5.8 Design rationale

Design rationale is information capturing the reasoning of the designer which led to the system as designed, including design options, tradeoffs considered, decisions made, and the justifications of those decisions.

Design rationale takes the form of commentary, made throughout the decision process and associated with collections of design elements. It captures the reasoning that led to the system as it has been designed. Design rationale includes: design issues raised and addressed in response to design concerns; design options considered; tradeoffs evaluated; decisions made; criteria used to guide design decisions; and arguments and justifications made to reach decisions.

5.9 Design languages

A *design language* is a notation, representational scheme or other modeling technique used to develop, analyze, and document a software design. There are many design languages used to describe software designs. Design languages are selected as a part of design viewpoint declaration (5.5).

A design language may be selected for a design viewpoint only if it supports all modeling elements defined by that viewpoint.

For use in SDDs, design languages shall be selected which have:
- a well-defined syntax and semantics; and
- the status of an available standard or equivalent defining document.

In a conforming SDD, only standardized and established* (defined and convenient) design languages shall be used. In the case of a newly-invented design language, the language definition must be provided as a part of the viewpoint declaration.

NOTE—Standardized design languages in common use are preferable to established one without a formal definition. Examples of standardized languages include: IDEF0, IDEF1X, Conceptual Graphs, UML, VDL, and Z. Examples of established languages include: Petri Nets, State Machines, Automata, Decision Tables, Warnier Diagrams, JSP, PDL, Structure Charts, HIPO, JSD, Reliability Models, Queueing Models.

NOTE—It is acceptable to use a design language in more than one view. It is also acceptable to use more than one design language within any of the views as long as they were declared by the viewpoint. Even for the portion of the design as long as one is used as a basis for interchange whenever that is necessary due to organizational consideration such as development by non-collocated team members, subcontracting of a partial design responsibility, taking advantages of several case tools and/or key designer's expertise, etc.

NOTE—Annex B establishes a uniform format for describing design languages to be used in SDDs.

NOTE—In case that no adequate design language is readily available for a specified viewpoint, it is the designer's responsibility to provide an adaptation of an existing language or the definition of an appropriate new design language. In exceptional cases, the definition provided by the designer shall be included in the SDD after proper evaluation for the reference before any use of such language is to be approved for capture of a view information in an SDD. This does not restrict exploratory use in sketching designs but excludes any use in formal documents, blueprints and executable specifications without prior update of SDD metadata with adequate design language. Informal documents without notification of nonstandard language or nonstandard use of a design language in them, can not be included in a conformant SDD. This note clarifies dynamic nature of leading edge design technology and points to a process necessary in adapting SDD metadata before populating or communicating SDD (design state) view information. Explicit status of not-yet-approved SDD information is necessary if some exploratory methodological information is to be included for review purposes and archived with rationale for decisions made.

6. Design Viewpoints

6.1 Introduction

This clause defines several design viewpoints for use in SDDs. It illustrates the specification of viewpoints in terms of design language selections, relates design concerns with viewpoints and establishes language-(notation-, method-, and process-) neutral names for selected viewpoints. Table 1 summarizes these viewpoints. For each viewpoint, its name, design concerns, and appropriate design languages, are listed. Short descriptions relating a minimal set of design entities, design relationships, design entity attributes, and design constraints are provided for each viewpoint. Additional references pertinent to the use of each viewpoint are also listed.

These viewpoints are required with a caveat, a qualified designer judgment is necessary to tailor out viewpoints not of interest in a particular situation, or to refine viewpoints (see for example viewpoint 2).

Table 6.1 also list several suggested Overlays. Furthermore it is an explicit requirement of this standard that designer has responsibility of determining additional (using her professional judgment) viewpoints and explicitly provides information on concerns (interested stakeholders), design elements, relationships, attributes and constraints of interest and select (define) appropriate design languages.

Design Viewpoint	**Design Concerns**	**Example Design Languages**
Context (6.2)	Systems services and users	IDEF0, UML Use Case Diagram, Structured Analysis Data Flow Context Diagram
Composition (6.3) *Can be refined into new viewpoints, such as: functional (logical) decomposition and run-time (physical) decomposition.*	Composition and modular assembly of systems in terms of subsystems and (pluggable) components, buy vs. build, reuse of components	*Logical*: UML Package Diagram, UML Component Diagram, Architecture Description Languages, IDEF0, Structure Chart, HIPO *Physical*: UML Deployment Diagram
Logical (6.4)	Static structure (Classes, Interfaces and their relationships) Reuse of Types and implementations (Classes, data types)	UML Class Diagram, UML Object Diagram
Dependency (6.5)		
Information (6.6) with Data Distribution Overlay and Physical Volumetric Overlay	Persistent Information	IDEF1X, UML Class Diagram, variety of ER Diagrams
Patterns (6.7)	Reuse of Patterns and available Framework Template	UML Collaboration Diagram
Interface (6.8)		
Structure (6.9)	Internal structure of components in terms of components and classes in terms of classes	UML Internal (composite) Structure Diagram, UML Class Diagram
Interaction (6.10)	Object Interaction,	UML Sequence Diagram, UML

	messaging	Communication Diagram
State Dynamics (6.11)	Dynamic state transformations	UML State Machine Diagram, Statechart Diagram (Harel's), State Transition Table (Matrix), Automata, Petri Net
Algorithm (6.12)	Procedural logic	Decision Table, Warnier Diagram, JSP, PDL, (pseudo) code C#, Java, etc.
Resources (6.13) *May be refined into resource based viewpoints with possible Overlays*	Resource utilization	UML RT Profile, UML Class Diagram, UML OCL

Table 1. Summary of Design Viewpoints

6.2 Context Viewpoint

The Context Viewpoint is used to depict the services provided by a design subject with reference to an explicit context. That context is defined by reference to actors which include users and other stakeholders which interact with the design subject in its environment. The Context Viewpoint provides a "black box" perspective on the design subject.

Services depict an inherently functional aspect or anticipated cases of use of the design subject (hence "use cases" in UML). Stratification of services and their descriptions in the form of scenarios of actors' interactions with the system provide a mechanism for adding detail. Services may also be associated with actors through information flows. The content and manner of information exchange with the environment implies additional design information and the need for additional viewpoints (e.g., Interaction Viewpoint).

A Deployment Overlay to a Context view can be transformed into a Deployment view whenever the execution hardware platform is part of the design subject; for stand-alone software design, a Deployment Overlay maps software entities onto externally available entities not subject of the current design effort. Similarly, work allocation to teams and other management perspectives are overlays in the design.

6.2.1 Design Concerns

The purpose of the Context Viewpoint is to identify a design subject's offered services, its actors (users and other interacting stakeholders), to establish the system boundary, to effectively delineate the design subject's scope of use, operation.

Drawing a boundary separating a design subject—whether system, subsystem, or component—from its environment, determining a set of services to be provided, and the information flows between design subject and its environment, is typically a key design decision; making this viewpoint applicable to most design efforts.

As the system is portrayed as a black box, with internal decisions hidden, the Context view is often a starting point of design, showing what is to be designed functionally as the only available information about the design subject: a name and an associated set of externally identifiable services. Requirements

analysis may identify these services with a specification of Quality of Service attributes, henceforth invoking many non-functional requirements. Frequently incomplete a context view is begun in requirements analysis and the work persists in completing this view during the design process.

6.2.2 View Elements

Entities: **Actors**: external active elements interacting with the design subject, including users, other stakeholders and external systems or other items. **Services**: also called use cases. Directed information **Flows** between the design subject, treated as a black box, and its actors associate actors with services. Flows capture the expected information content exchanged.

Relationships: receive generated output and provide received input (between actors and the design subject).

All design entities of this viewpoint are recursively decomposable into like entities to support hierarchical description. Therefore composition and generalization relationships are needed.

Constraints: Qualities of service, formats and media of interaction (provided to and received from) with environment as required by the environment are design constraints for this viewpoint.

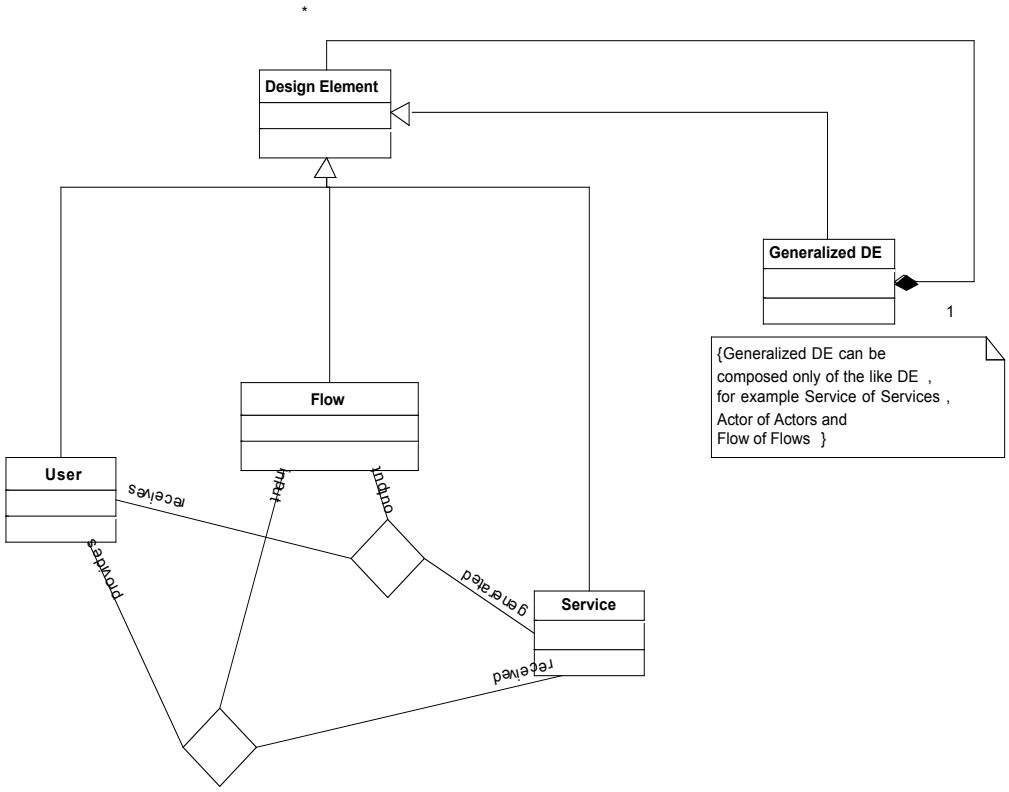


Figure 3. Meta model of Context Viewpoint

6.2.3 Example languages

Any “black box”-type diagrams can be used to realize the Context Viewpoint. Appropriate languages include data flow languages for Structured Analysis (e.g., IDEF0 or those of the DeMarco or Gane-Sarson variety), the Cleanroom’s Black Box Diagrams and UML use cases.

6.3 Composition Viewpoint

The Composition Viewpoint describes the way the design subject, as an evolving solution, is (recursively) structured into constituent large-grained parts and establishes the roles of those parts.

6.3.1 Design concerns

Software developers and maintainers use this viewpoint to identify the major design constituents of the design subject, to localize and allocate functionality, responsibilities, or other design roles to these constituents. In maintenance, it can be used to conduct impact analysis and localize the efforts of making changes. Reuse, on the level of existing subsystems and large-grained components, can be addressed as well. The information in a composition view can be used by acquisition management and in project management for specification and assignment of work packages, and for planning, monitoring, and control of a software project. This information, together with other project information, can be used in estimating cost, staffing, and schedule for the development effort. Configuration management may use the information to establish the organization, tracking, and change management of emerging work products [IEEE Std 828-1998].

6.3.2 View elements

Entities: The design entities are the types of constituents of a system: **subsystems**, **components**, **modules**, **ports**, and (provided and required) **interfaces**. Entities of interest include also **libraries, frameworks, software repositories** and **catalogs, templates,** and independent **functions**.

Relationships: The key design relationships are **composition, use** and **generalization**. The Composition Viewpoint supports the recording of the part-whole relationships between these entities using **realization, dependency, aggregation, composition and generalization** relationships. Additional design relationships are **required** and **provided** (for interfaces), and the **attachment** of ports to components.

Attributes: For each entity, the viewpoint provides a reference to the detailed description via the **identification** attribute. The attribute descriptions for **identification**, **type**, **purpose**, **function**, and **definition** attribute should be included in this design view.

6.3.2.1 Function attribute

A statement of what the entity does. The **function** attribute shall state the transformation applied by the entity to inputs to produce the desired output. In the case of a data entity, this attribute shall state the type of information stored or transmitted by the entity.

NOTE—This design entity attribute is retained for compatibility with IEEE 1016–1998.

6.3.2.2 Subordinates attribute

The identification of all entities composing this entity. The **subordinates** attribute shall identify the composed-of relationship for an entity. This information is used to trace requirements to design entities and to identify parent/child structural relationships through a software system decomposition.

NOTE—This design entity attribute is retained for compatibility with IEEE 1016–1998. Equivalent capability is available through the **composition** relationship.

6.3.3 Example languages

UML Component Diagrams cover this viewpoint. The simplest graphical technique used to describe functional system decomposition is a hierarchical decomposition diagram, such diagram can be used together with natural language descriptions of purpose and function for each entity. One example is provided by IDEF0 the other options are Structured Chart, and IBM's HIPO Diagram. Run time composition representation can also use Structured Diagrams.

6.4 Logical Viewpoint

The purpose of the Logical Viewpoint is to elaborate existing and designed types and their implementations as classes and interfaces with their structural static relationships. This viewpoint also uses examples of instances of types in outlining design ideas.

6.4.1 Design Concerns

The Logical Viewpoint is used to address the development and reuse of adequate abstractions and their implementations. For any implementation platform, a set of types is readily available for the domain abstractions of interest in a design subject and a number of new types is to be designed, some of which may be considered for reuse. The main concern is the proper choice of abstractions and their expression in terms of existing types (some of which may had been specific to the design subject).

6.4.2 View elements

Entities: **class**, **interface**, **power type**, **data type**, **object**, **attribute**, **method**, **association class**, **template**, and **namespace**.

Relationships: **association, generalization, dependency, realization, implementation, instance of, composition,** and **aggregation**.

Attributes: **name, role name, visibility, cardinality, type, stereotype, redefinition, tagged value, parameter**, and **navigation efficiency**.

Constraints: **value constraints, relationships exclusivity constraints, navigability, generalization sets, multiplicity, derivation, changeability, initial value, qualifier, ordering, static, precondition, postcondition,** and **generalization set constraints**.

6.4.3 Example languages

UML Class Diagrams and UML Object Diagrams (showing objects as instances of their respective classes). Lattice of types, and references to available implemented types are commonly used as supplementary information for this viewpoint.

6.5 Dependency Viewpoint

The Dependency Viewpoint specifies the relationships of interconnection and access among entities. These relationships include shared information, prescribed order of execution, or well-defined parameterized interfaces.

6.5.1 Design concerns

A dependency view provides an overall picture of the design subject in order to assess the impact of requirements or design changes. It can help maintainers to isolate entities causing system failures or

resource bottlenecks. It can aid in producing the system integration plan by identifying the entities that are needed by other entities and that must be developed first. This description can also be used by integration testing to aid in the production of integration test cases.

6.5.2 View elements

Entities: **subsystem**, **component** and **module**. *Attributes*: **identification**, **type**, **purpose**, **dependencies**, and **resources**. This attribute information should be provided for all design entities. *Relationships*: **uses**, **provides** and **requires**.

6.5.2.1 Dependencies attribute

A description of the relationships of this entity with other entities. The dependencies attribute shall identify the uses or requires the presence of relationship for an entity. These relationships are often graphically depicted by structure charts, data flow diagrams, and transaction diagrams.

This attribute shall describe the nature of each interaction including such characteristics as timing and conditions for interaction. The interactions may involve the initiation, order of execution, data sharing, creation, duplicating, usage, storage, or destruction of entities.

NOTE—This design entity attribute is retained for compatibility with IEEE 1016–1998.

6.5.3 Example Languages

There are a number of methods that help minimize the relationships among entities by maximizing the relationship among elements in the same entity. These methods emphasize low module coupling and high module cohesion [Freeman and Wasserman].

UML component diagrams and UML package diagrams showing dependencies among subsystems [OMG, UML].

6.6 Information Viewpoint

The Information Viewpoint is applicable when there is a substantial persistent data content expected with the design subject.

6.6.1 Design concerns

Key concerns include: persistent data structure, data content, data management strategies, data access schemes, and definition of metadata.

6.6.2 View elements

Elements: **data items**, **types** and **classes**, **data stores**, **access mechanisms**. *Attributes*: persistence and quality properties. *Relationships*: **association**, **uses**, **implements**. Data attributes, their constraints and static relationships among data entities, aggregates of attributes and relationships.

6.6.2.1 Data attribute

A description of data elements internal to the entity. The data attribute shall describe the method of representation, initial values, use, semantics, format, and acceptable values of internal data.

The description of data may be in the form of a data dictionary that describes the content, structure, and use of all data elements. Data information shall describe everything pertaining to the use of data or internal data

structures by this entity. It shall include data specifications such as formats, number of elements, and initial values. It shall also include the structures to be used for representing data such as file structures, arrays, stacks, queues, and memory partitions.

The meaning and use of data elements shall be specified. This description includes such things as static versus dynamic, whether it is to be shared by transactions, used as a control parameter, or used as a value, loop iteration count, pointer, or link field. In addition, data information shall include a description of data validation needed for the process.

NOTE—This design entity attribute is retained for compatibility with IEEE 1016–1998.

6.6.3 Example Languages

IDEF1X, UML Class Diagrams, a variety of ER-type diagrams.

6.7 Patterns Use Viewpoint

This viewpoint addresses design ideas (emergent concepts) as collaboration patterns involving abstracted roles and connectors.

6.7.1 Design concerns

Key concerns include: reuse at the level of design ideas (design patterns), architectural styles and framework templates.

6.7.2 View elements

Entities: **collaboration**, **class**, **connector**, **role**, **framework template**, **pattern**. *Relationships*: **association**, **collaboration use**, **connector**. *Attributes*: **name**. *Constraints*: collaboration constraints.

6.7.3 Example languages

UML Collaboration Diagram and a combination of the UML Class Diagram and the UML Package Diagram.

6.8 Interface Viewpoint

The Interface Viewpoint provides information designers, programmers, and testers the means to know how to correctly use the services provided by a design subject. This description includes the details of external and internal interfaces not provided in the software requirements description. This viewpoint consists of a set of interface specifications for each entity.

NOTE—User interfaces are addressed separately.

6.8.1 Design concerns

An interface view description serves as a binding contract among designers, programmers, customers, and testers. It provides them with an agreement needed before proceeding with the detailed design of entities. In addition, the interface description may be used by technical writers to produce customer documentation or may be used directly by customers. In the latter case, the interface description could result in the production of a human interface view.

Designers, programmers, and testers may need to use design entities that they did not develop. These entities may be reused from earlier projects, contracted from an external source, or produced by other

developers. The interface description settles the agreement among designers, programmers, and testers about how cooperating entities will interact. Each entity interface description should contain everything another designer or programmer needs to know to develop software that interacts with that entity. A clear description of entity interfaces is essential on a multi-person development for smooth integration and ease of maintenance.

6.8.2 View elements

The attribute descriptions for identification, function, and interfaces should be included in this design view. This attribute information should be provided for all design entities.

6.8.2.1 Interface attribute

A description of how other entities interact with this entity. The interface attribute shall describe the methods of interaction and the rules governing those interactions. The methods of interaction include the mechanisms for invoking or interrupting the entity, for communicating through parameters, common data areas or messages, and for direct access to internal data. The rules governing the interaction include the communications protocol, data format, acceptable values, and the meaning of each value.

This attribute shall provide a description of the input ranges, the meaning of inputs and outputs, the type and format of each input or output, and output error codes. For information systems, it should include inputs, screen formats, and a complete description of the interactive language.

NOTE—This design entity attribute is retained for compatibility with IEEE 1016–1998

6.8.3 Example Languages

The interface description should provide the language for communicating connections, content expected, so that the interface can serve as a contract for cooperating design elements. Interfaces as connectors may be treated as first order design entities.

In case of GUI interfaces the UI description should provide language for communicating with each entity to include screen formats, valid inputs, and resulting outputs. For those entities that are data driven, a data dictionary should be used to describe the data characteristics. Those entities that are highly visible to a user and involve the details of how the customer should perceive the system should include a functional model, scenarios for use, detailed feature sets, and the interaction language.

6.9 Structure Viewpoint

The Structure Viewpoint is used to document the internal structure of coarse-grained components and classes in terms of like elements (recursively).

6.9.1 Design concerns

Compositional structure of coarse-grained components and classes and reuse of fine-grained components and classes.

6.9.2 View elements

Entities: **port**, **connector**, **interface**, **part**, **class**. *Relationships*: **connected**, **part of**, **enclosed**, **provided**, **required**. *Attributes*: **name**, **type**, **purpose**, **definition**. *Constraints*: interface constraints, reusability constraints, dependency constraints.

6.9.3 Example languages

UML Internal (composite) Structure Diagram, UML Class Diagram and UML Package Diagram.

6.10 Interaction Viewpoint

The Interaction Viewpoint defines strategies for interaction among entities, provides information needed to perceive how, why, where, and at what level actions occur. This could include designing with concurrent tasks and/or asynchronous messaging, messaging among objects, etc.

6.10.1 Design concerns

Designers for responsibility allocation in collaborations especially when adapting design patterns. Discovery of or detail description of interactions in terms of messages (method calls) among affected objects in fulfilling required actions.

For designing state transition logic and concurrent tasks, for reactive, interactive, distributed, real-time, and similar systems.

6.10.2 View elements

Classes, methods. states, events, signals, hierarchy, concurrency, timing and synchronization.

6.10.3 Examples

6.11 State Dynamics Viewpoint

Reactive systems and systems whose objects may have interested states require this dynamic viewpoint.

6.11.1 Design concerns

Modes, states, transitions, and constrains in time ordered events- systems dynamic

6.11.2 View elements

Entities: **event**, **condition**, **state**, **transition**, **activity**, **composite state**, **submachine state**, **region**, **trigger**. *Relationships*: **part-of**, **internal**, **effect**, **entry**, **exit**, **attached-to**. *Attributes*: **name**, **completion**, **active**, **initial**, **final**. *Constraints*: guard conditions, concurrency, synchronization, state invariant, transition constraint, protocol.

6.11.3 Example languages

UML State machine Diagrams, Statechart Diagram (Harrel's), State Transition Table (Matrix), Automata, Petri Net.

6.12 Algorithm Viewpoint

The detailed design description of operations (methods, functions), the internal details, logic, of each design entity; this applies to components, classes, and individual methods as design entities.

6.12.1 Design concerns

This description contains the details needed by programmers, analysts of algorithms re time-space performance and specifically coders prior to implementation. The detailed design description can also be used to aid in producing unit test plans.

6.12.2 View elements

These details include the attribute descriptions for identification, processing, and data. This attribute information should be provided for all design entities.

6.12.1 Processing attribute

A description of the rules used by the entity to achieve its function. The processing attribute shall describe the algorithm used by the entity to perform a specific task and shall include contingencies. This description is a refinement of the function attribute. It is the most detailed level of refinement for this entity.

This description should include timing, sequencing of events or processes, prerequisites for process initiation, priority of events, processing level, actual process steps, path conditions, and loop back or loop termination criteria. The handling of contingencies should describe the action to be taken in the case of overflow conditions or in the case of a validation check failure.

NOTE—This design entity attribute is retained for compatibility with IEEE 1016–1998.

6.12.3 Examples

Decision tables, flowcharts, actual programming languages may also be used.

6.13 Resource Viewpoint

The purpose of the Resource Viewpoint is to model the characteristics and utilization of resources in a software design.

6.13.1 Design concerns

Key concerns include: resource utilization, availability, performance.

6.13.2 View elements

Entities: Resources, usage policies, performance measures. *Relationships*: Allocation. *Attributes*: resource name, rate of consumption, units of measurement. *Constraints*: Priorities, Locks, Resource Constraints,

6.13.2.1 Resources attribute

A description of the elements used by the entity that are external to the design. The resources attribute shall identify and describe all of the resources external to the design that are needed by this entity to perform its function. The interaction rules and methods for using the resource shall be specified by this attribute.

This attribute provides information about items such as physical devices (printers, disc-partitions, memory banks), software services (math libraries, operating system services), and processing resources (CPU cycles, memory allocation, buffers).

The resources attribute shall describe usage characteristics such as the process time at which resources are to be acquired and sizing to include quantity, and physical sizes of buffer usage. It should also include the identification of potential race and deadlock conditions as well as resource management facilities.

NOTE—This design entity attribute is retained for compatibility with IEEE 1016–1998.

6.13.3 Examples

UML RT Profile, UML Class Diagram, UML OCL.

==========

Annex A (informative)
Bibliography

Abran, A. and J.W. Moore, *Guide to the Software Engineering Body of Knowledge: 2004 Edition*. IEEE Press, 2005.

Arlow J., and I. Neustadt, *Enterprise Patterns and MDA: Building better software with archetype patterns and UML*. Addison-Wesley, 2004.

Coplien J., *Multi-Paradigm Design for C++*. Addison-Wesley, 1998.

D'Souza D., and A. Wills, *Objects, Components, and Frameworks with UML*. Addison-Wesley 1999.

Douglass B.P., *Real-Time UML*. 3rd edition, Addison-Wesley, 2004.

Douglass B.P., *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Addison-Wesley, 1999.

Draft ISO Standard *Conceptual Graphs*. 2001 (See: http://www.jfsowa.com/cg/cgstand.htm).

Evitts P., *A UML Pattern Language*. Sams, 2000.

Fayad, M.E., and R.E. Johnson (editors), *Domain-Specific Application Frameworks*. Wiley, 2000.Feldmann C.G., *The Practical Guide to Business Process Reengineering Using IDEF0*. Dorset House Publishing, 1998.

Gamma E. et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Gomma H., *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, 2000.

Gomma H., *Designing Software Product Lines with UML*. Addison-Wesley, 2005.

Graham I., *Object-Oriented Methods*. Addison-Wesley, 2001.

IEEE Std 1012–1998, *IEEE Standard for Software Verification and Validation*.

IEEE Std 1016–1998, *IEEE Recommended Practice for Software Design Descriptions*.

IEEE Std 1320.1–1998, *IEEE Standard for Functional Modeling Language—Syntax and Semantics for IDEF0*.

IEEE Std 1320.2–1998, *IEEE Standard Conceptual Modeling Language—Syntax and Semantics for IDEF1X*.

IEEE Std 1471–2000, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*.

IEEE Std 830–1998, *IEEE Recommended Practice for Software Requirements Specifications*.

IEEE/EIA Std 12207.1–1997, *Industry Implementation of International Standard ISO/IEC 12207:1995 Software life cycle processes—Life cycle data*.

IEEE/EIA Std 12207.2–1997, *Industry Implementation of International Standard ISO/IEC 12207:1995 Software life cycle processes—Implementation considerations*.

ISO/IEC 13568:2002, *Information technology—Z formal specification notation—Syntax, type system and semantics*.

ISO/IEC 13817–1:1996, *Information technology—Programming languages, their environments and system software interfaces—Vienna Development Method—Specification Language—Part 1: Base language*.

Kruchten, P. The Rational Unified Process. Addison-Wesley, 2000.

OMG formal/05-04-01, *Unified Modeling Language (UML)*. version 1.4.2, 2001.

OMG formal/05-07-04, *UML 2.0 Superstructure Specification*.

Page-Jones M., *The Practical Guide to Structured Systems Design*. second edition, Prentice Hall, 1988.

Parnas, D.L. and P.C. Clements, A rational design process: how and why to fake it. *IEEE Transactions on Software Engineering*, **12**(7), 1986.

Ross D.T., J.B. Goodenough and C.A. Irvine, Software engineering: Process, principles, and goals. *COMPUTER* **8**(5) (May 1975): 17–27

Ross D.T., Structured Analysis: a language for communicating ideas. *IEEE Transactions on Software Engineering*, 1977.

Rumbaugh J., I. Jacobson, and G. Booch, *UML Reference Manual*. second edition, Addison-Wesley, 2005.

Shon D., *The Reflective Practitioner*. Basic Books, 1983.

Weiss D. and C. Lai, *Software Product Line Engineering*. Addison-Wesley, 1999.

Winograd T., (editor). *Bringing Design to Software*. Addison-Wesley, 1996.

Yourdon E., and L. Constantine, *Structured Design*. Prentice Hall, 1979.

Annex B (informative)
Conforming Design Language Description

NOTE TO REVIEWERS: At the 15-16 February 2003 meeting in Savannah, GA, it was proposed that it would be useful to have a "standard" template for exchanging information about design languages conforming to P1016. XML would be one useful way to describe such a template. It was further proposed that this would be a more useful annex to P1016 than the current planned summaries of IDEF0, IDEF1 and UML. The working group investigated the P1016 PAR [charter with the IEEE Software Engineering Standards Committee] to see whether this change would impact our planned work, or would require a PAR revision. The only relevant text in the PAR (18 Oct 2001 version) was to "harmonize with IEEE Stds 1302 (IDEF0 and IDEF1). This would be easily accommodated with the new work plan. It was further suggested that this proposal would generate useful interest among users and vendors to apply the terminology and concepts of the standard in describing their design language(s).

The group agreed to draft text outlining this change to place into the next revision (D3.0) and prepare an annex to replace current annex drafts (IDEF, UML) with an XML DTD. This is that annex.

This annex defines a uniform format for describing design languages. Any design language may be documented in terms of a number of characteristics. These characteristics have been chosen to facilitate the selection of design languages in the selection and definition of viewpoints (clause 5). The format has a textual form intended to allow both human and machine-readable application.

It is envisioned the providers of design languages (whether commercial, industrial, research or experimental) will be able to document the intended usage of the design language using this format. This documentation will allow Designers to more readily review the properties of that design language for use in an SDD because the attributes captured in the DLD match the considerations to be made when defining a design viewpoint in accordance with clauses 5 and 6 of this standard.

B.1 Information on Conforming Design Languages (normative?)

Every well-formed design language description must specify the design language name.

Examples: IDEF0, UML StateChart

Every well-formed design language description must have a reference to the definition of the design language. This may be a reference to a standard or other defining document.

Examples: IEEE Std 1302.1, OMG-Unified Modeling Language, v1.4 September 2001

Every well-formed design language description must contain an identification of one or more design concerns which are capable of being expressed using this design language. This information can be used by Designers to choose appropriate design languages to implement selected design viewpoints within an SDD.

Examples: Functionality, Reliability

Every well-formed design language description must identify each design entity type defined by the design language.

Examples: State, Transition, Event

Every well-formed design language description must identify each design entity attribute type, and the design entity that define it.

Examples: transitionLabel *defined by*: Transition; guardCondition *defined by*: Transition.

Every well-formed design language description must identify the design entity relationship types that are a part of the design language. First, the relationship type is named. Then the design entity types participating in the relationship are identified.

Examples: subActivities participants: 2 or more Activities

B.2 Example: StateCharts

Design Language Name:

Design Language Source:

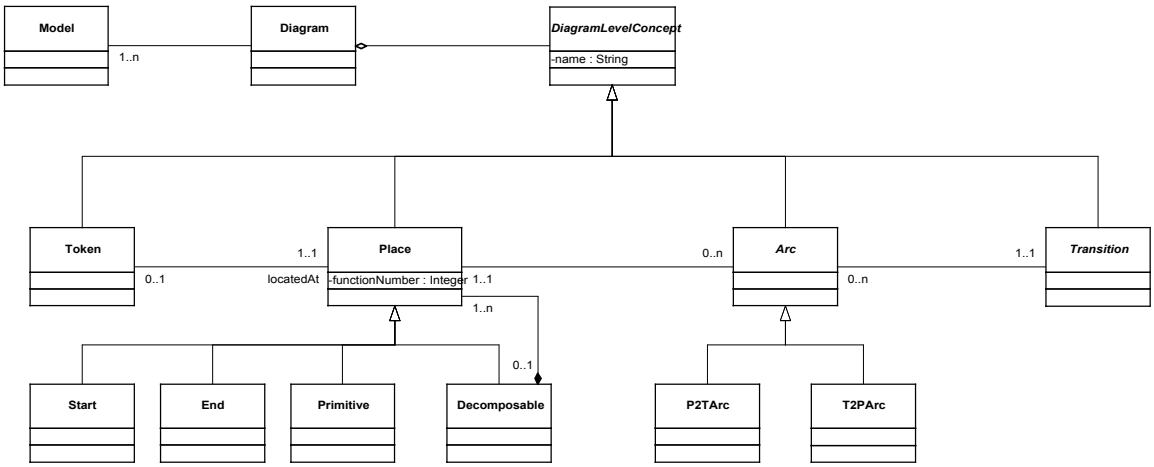
Expressible Concerns:

Design Entities:

Design Entity Attributes:

Desgin Relationships:

PetriNet metamodel



B.3 Example: IDEF0

Design Language Name:

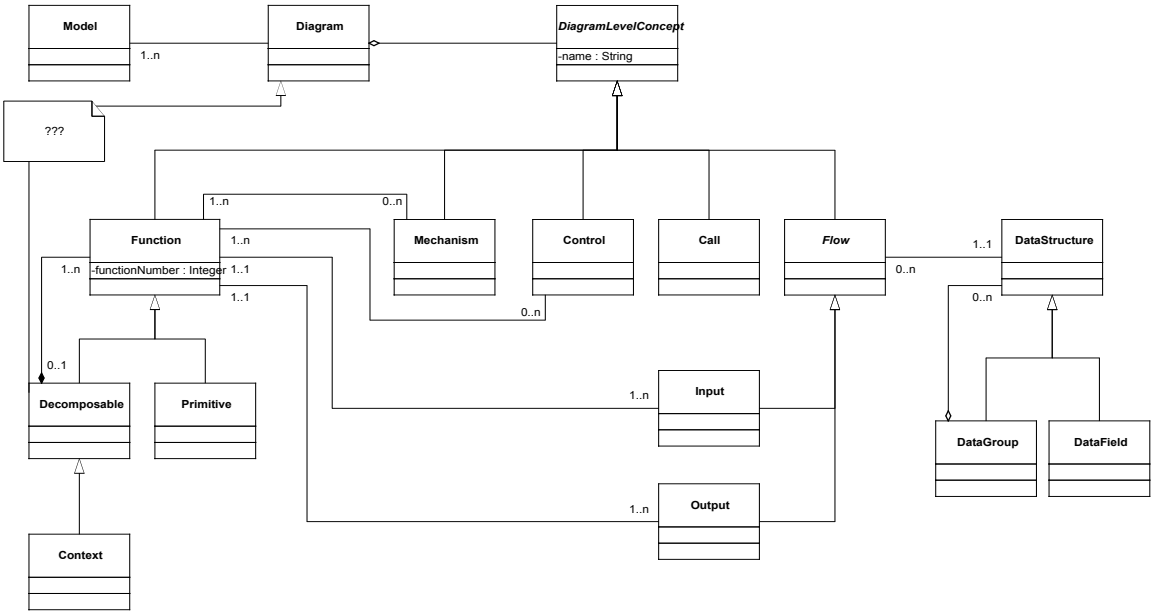
Design Language Source:

Expressible Concerns:

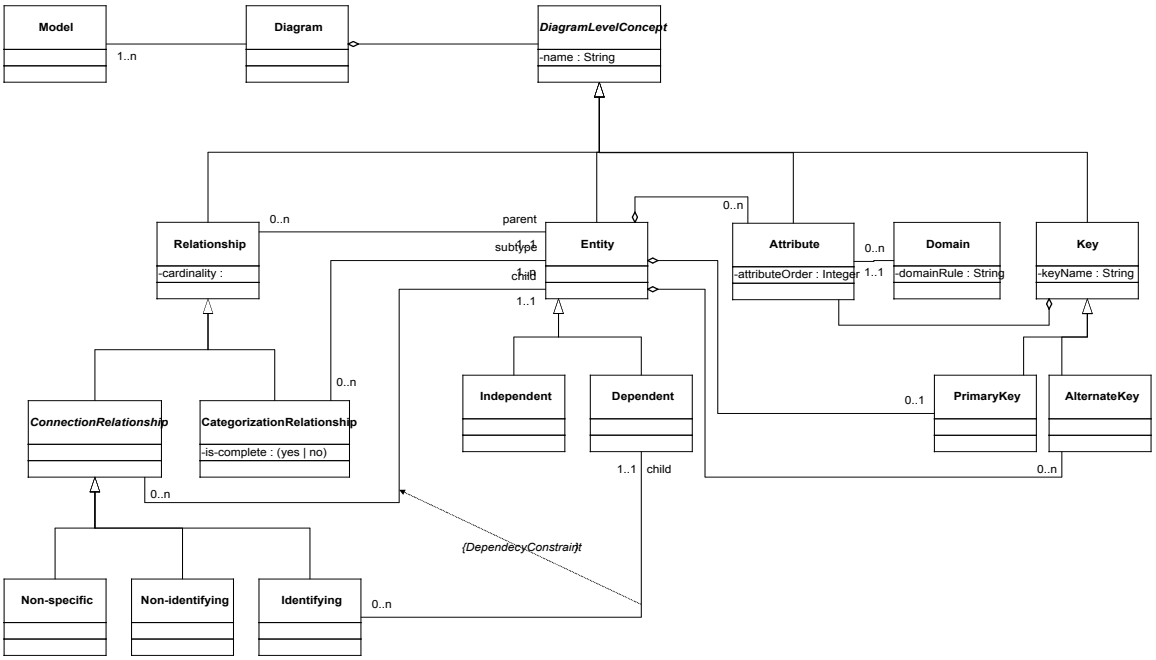
Design Entities:

Design Entity Attributes:

Design Relationships:



B.4 Example: IDEF1



Annex C (informative)
Templates for an SDD

The following templates show some possible ways to organize and format an SDD conforming to the requirements of Clause 5.

Frontspiece
- Date of Issue and Status
- Issuing organization
- Authorship
- Change history

Introduction
- Purpose
- Scope
- Context
- Summary

References

Glossary

Body
- Identified Stakeholders and Design Concerns
- Design Viewpoint 1
- Design View 1
- Design Viewpoint 2
- Design View 2
- Etc.

Figure C.1—Table of contents for an SDD