Chapter 19: Software Design Process

The software design process translates the requirement specification what into another phase of the software development life cycle: how to develop the software. The software design is a blueprint and road map of software development. The software design process allocates the functional requirements to a design structure. This chapter discusses the software design architecture; software design methods, including the object−oriented design method and the structured design methods; and software design descriptions.

Software Design Architecture

The software design architecture represents the physical design of a software system. The architecture primarily captures static information. A model contains one or more architecture diagrams. Multiple architecture diagrams group meaningful collections of objects. Each architecture diagram contains symbols that denote components and dependencies. The components represent structural elements that are provided by the underlying implementation language. Dependencies represent compilation dependencies among components. The symbols for an architecture diagram are somewhat language specific because not all languages provide the same packaging mechanisms, generic units, and tasks.

An architecture diagram takes the form of a graph in which components are vertices of the graph, dependencies are directed arcs, and the topology of the graph satisfies the rules of separate compilation for the programming language.

For each component, a name, its semantics, and relevant design notes are provided. Recording the kind of components that use the vocabulary of the underlying implementation language captures static design decisions. Each component references a set of classes from class structures of the same project model or objects from object diagrams of the same project model. This set of entities registers the design decisions with regard to packaging of the implementation of individual classes and objects. Typically, a one−to−one mapping of classes to components and many−to−one mappings of objects to components exist.

If the rules of underlying implementation language allow, some components explode and reveal the unveiling of another complete architecture diagram. Dependencies denote asymmetric relationships among components.

Production of a diagram motivates changes in corresponding class structures and object diagrams because the logical design is modeled to the physical constraints of the system. These diagrams facilitate the representation and sharing of components among programs, even when such programs execute in a distributed environment.

What is Software Design?

The software design process converts the software requirements analysis into a logical design for the computer software configuration item (CSCI) (Figure 19−1). The software developers further decompose and allocate the requirements to the proper functionality associated components. They construct a hierarchy chart to represent the partitioning of complex functions into simpler functions. They refine software components into lower levels, which contain software units that can be coded, compiled, and tested. The software developers ensure that all of the requirements are differentiated properly and allocated from software components to software units.

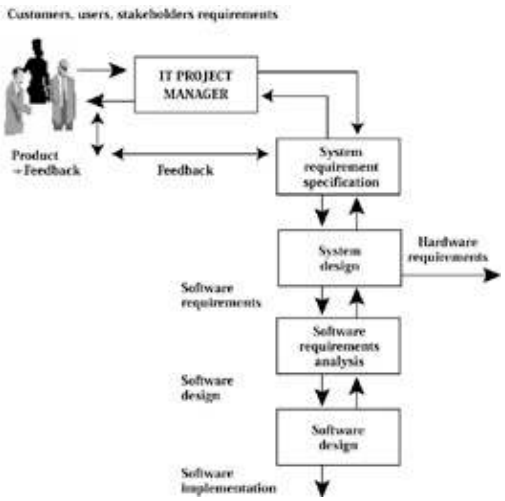

Figure 19−1: IT software design process

The software developers develop a detailed design for the interface external to the software item, between software components, and between the software units. The software requirements are traceable forward and backward. The software developers also develop and document a detailed design for the database.

The software developers define and document test requirements and a schedule for testing software units. The test requirements include stressing the software unit at the limits of its requirements. The software developers update the test requirements and the schedule for software integration. The software developers evaluate the software detailed design and test requirements for the following:

- Traceability to the requirements of the software item
- External consistency with architectural design
- Internal consistency between software components and software units
- Appropriateness of design methods and standards used
- Feasibility of testing
- Feasibility of operation and maintenance

Software Design Methods

Software design methods cover a range of abstraction. A design method includes the choice of environments and approaches to be used for software design. The strategy includes techniques that develop software efficiently. The principal software design methods are the object−oriented method and the structured design method.

Object−Oriented Method

The object−oriented method (OOM) is the combination of an object−oriented approach, a data structure approach with entity−relationship modeling, and a functional approach. The goals of the OOM are to capture in detail the domain−specific knowledge of the application in a form that lends itself to careful point−by−point verification by the domain experts.

Through formal models of the problem domain, the OOM provides a detailed and well−documented foundation on which the software developers make requirement decisions. The method transfers the domain knowledge accurately to the software engineers and communicates the requirement analysis in a form that is easily understood and mapped into an object−oriented design.

OOM concepts understand the customers requirements, graphically show the logic to the customers, understand the stated requirements, determine objects, establish relationships between objects, determine instantiation criteria for objects and their relationships, and develop functional processes.

An object is a mental abstraction of a set of real world things. This method uses concepts from object−oriented and structured approaches, which includes abstract data types, inheritance, and module coupling and cohesion. It uses graphic models with proper documentation, which transfers the requirements from one phase to another for implementation. The OOM couples its analysis into object−oriented design (OOD) and is implemented in any suitable object−oriented programming language. This concept maintains the traceability of the requirements for embedded systems. The method provides the project controls and communication tools for management, quality assurance (QA), and documentation formats. The following are the basic OOM models:

- Object analysis model (OAM)
- Object information model (OIM)
- Object behavior model (OBM)
- Object process model (OPM)

Object Analysis Model

The OAM consists of all of the analysis that is needed so that the developers can understand the customers requirements. It also contains information with regard to all of the identified external interfaces. This step is especially important for embedded systems. The developers link these interfaces with external systems as illustrated in Figure 19−2. The concept of drawing this graphic is so that the customers requirements are understood in an unambiguous method and the customer understands his or her stated requirements. Identification of processes, data storage, and data flows can extend the model.

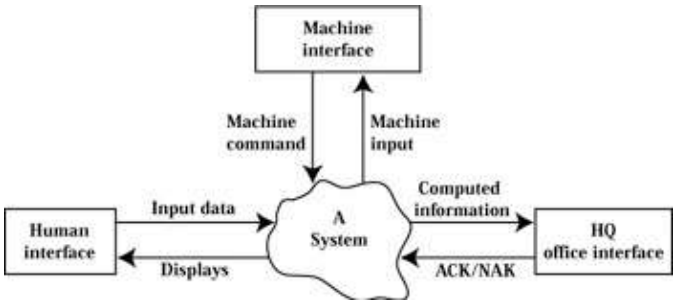

Figure 19−2: An OAM diagram

The goal is that the developers mutually understand and record the requirements in the proper document before proceeding to further analysis or design activity. The developers should clearly define the definition of the notations that are used in the graphic and properly record them in the document for understandability. They can initiate the object data dictionary (ODD) at this level by recording this essential information.

Object Information Model

The OIM identifies the conceptual entities of the problem and formalizes them as objects and attributes. Complete and unambiguous understanding of the problem is the goal. The OIM is the beginning of the design phase. It identifies things about the problem and their relationship. Things and associations are modeled here. The model is simple enough to be easily read and understood. Significant emphasis is placed on formalization of the relationships between objects.

The developers develop a model and depict it graphically as shown in Figure 19−3. They use the textual

descriptions in the definition of the models semantics. They group like things together into sets (see Figure 19−3). Things are alike if they behave in the same way and can be described by the same characteristics.

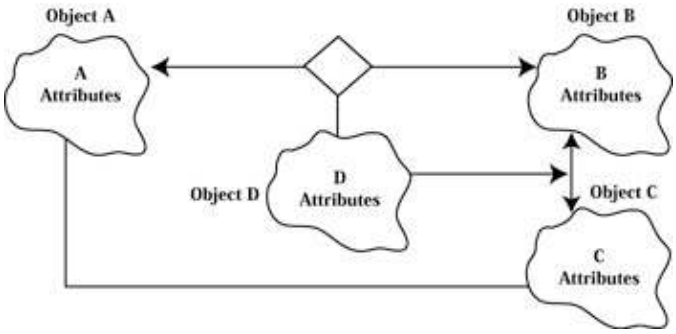

Figure 19−3: An OIM diagram

The characteristics that all elements of the set have in common are called *attributes* of the object. An attribute is the abstraction of a single characteristic, which is possessed by all entities and was abstracted as an object. The set of attributes must be complete, fully factored, and mutually independent.

The following is an example of the dot notation for attribute:

Object Attribute

The types of attributes are descriptive, naming, and referential. The descriptive attributes provide facts that are intrinsic to each instance of the object. The naming attribute provides a fact about the arbitrary name carried by each instance of the object. The referential attribute links an instance of one object to an instance of another object.

The developers use tables to define the types of questions that can be answered by inspection of the objects. The table name is the name of the object. Each column of the table is an attribute of the object, and each row is an instance of the object. The instance is the specific element of the set and is denoted by that object name. As illustrated in Figure 19−4, every box in a table contains exactly one value. Attributes of an object should not contain internal structure. Tables are the basic formal structure of the OIM because they are simple and adequate. The object representation by table assists in the identification of the objects and attributes. The table also helps in the representation of instances of objects.

Object A

Attribute A1	Attribute A2	Attribute A3
Instance - A		

Figure 19−4: OIM table

Associations exist between things in the real world. These associations must be formalized in this model. These can be recognized by verb phrases in the descriptions. For example:

The car *has* tires.

Thus a relationship is named by a verb phrase. It can be phrased in both directions of the relation. For example:

A class *is composed* of students.

or

Students *comprise* a class.

The same two objects may have more than one relationship between them, depending on their type of relationship. An object may be related to itself. A relationship can also exist between multiple objects. For example:

The floppy disk *was formatted* on the disk drive.

The floppy disk *is owned* by a student.

The floppy disk *contains* disk files.

This type of relationship, which involves two objects, can be classified into three fundamental forms and called *multiplicity:*

Multiplicity	Notation	Notation	Example
One−to−one		1:1	The husband has a wife.
One−to−many		1:M	The student owns books.
Many−to−many		M:M	The capacitor is a component of a computer.

A graphic notation is necessary for the replacement of a table for complex objects (Figure 19−5). The graphic notation eases handling of complex systems and represents spatial compression of model. Figure 19−6 illustrates OIM notation representations. Figure 19−7 shows supertype and subtype constructs. Figure 19−8 illustrates the correlation relationship symbol.

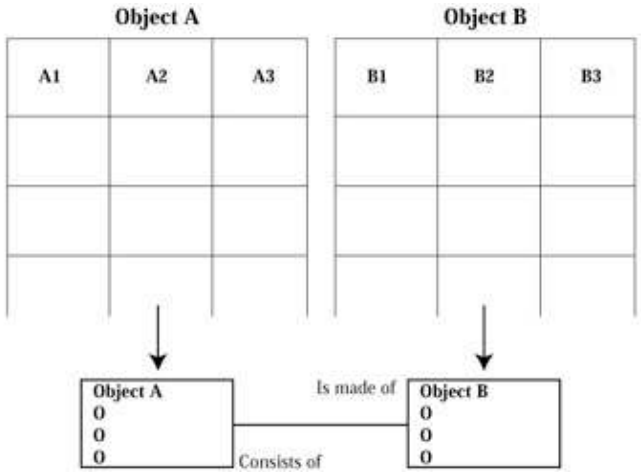

Figure 19−5: OIM table and object relationship

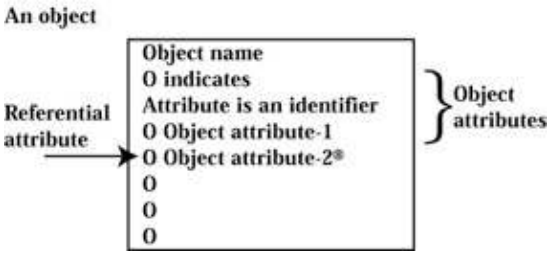

Figure 19−6: OIM notations

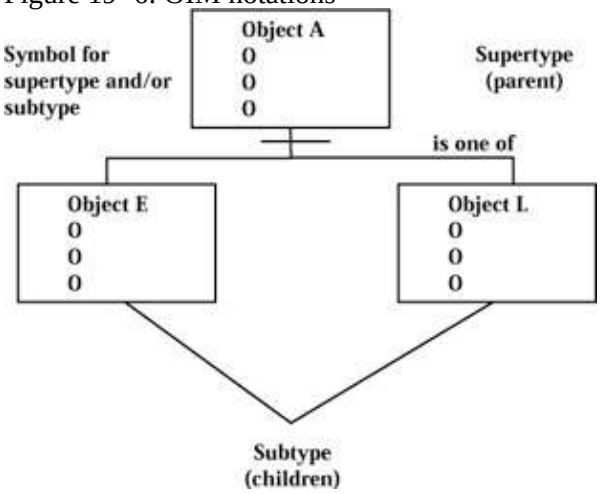

Figure 19−7: Supertype and/or subtype constructs

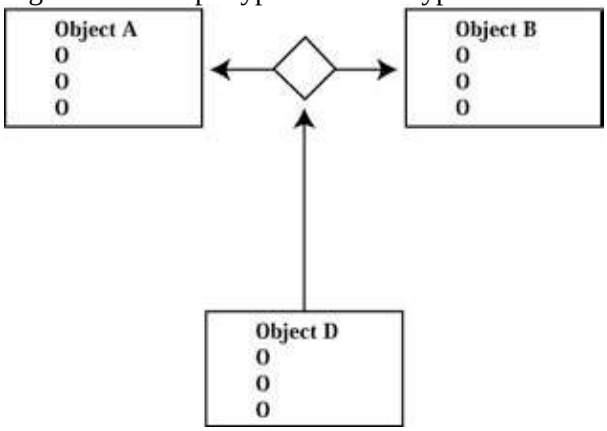

Figure 19−8: Correlation symbol

Object data dictionary. The developers document the object, attributes, and their relationships in an ODD. This document consists of a written description of objects, formalizes the identification of objects, forms part of the formal system specification, and separates descriptions that are written for each object. This is a live document and should start at the beginning of the software engineering phase and be enhanced throughout the software development and maintenance phases.

Selection criteria for a correct object. Four major tests help by not accepting false objects:

1. ***The Uniformity Test*** is based on the definition of an object, and each instance of the object must have the same set of characteristics and be subject to the same rule.
2. ***The More−Than−a−Name Test*** is applicable to objects that cannot be described by attributes. This object has no characteristics other than its name and is probably an attribute of another object.
3. ***The Or Test*** is conducted on the object description. If the inclusion criteria in the object description uses the word or in a significant manner, a set of diverse things is present rather than an object.

4. ***The More−than−a−List Test*** is conducted on the object description. If the inclusion criteria are in the object description, then it is simply a list of all specific instances of the object and is most likely not a true object.

Object Behavior Model

The OBM formalizes the life or event histories of objects and relationships as was identified in the OIM. Things go through various stages during their lifetime in the real world. The life cycle of an object is therefore the behavior of an object during its lifetime. Figure 19−9 presents an example of the object lifetime diagram.

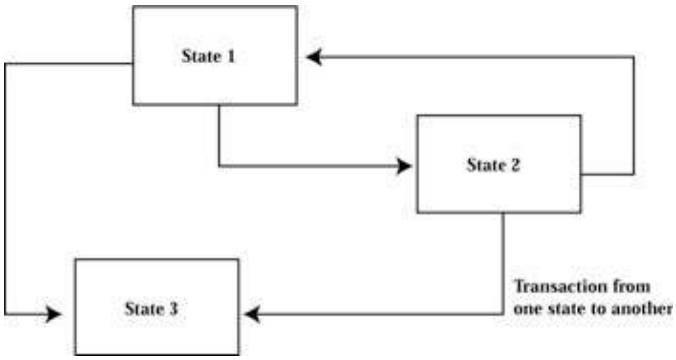

Figure 19−9: Object lifetime diagram

An object may be in only one stage at a time. The stages are mutually exclusive, they are discrete, and transitions can occur instantaneously. Transitions from any stage to any other stage are not always allowed. Incidents cause transition of things between stages. Some incidents cause a progression only when the thing is in certain stages of its life cycle. Similar characteristics of real world things have a common life cycle. Thus the OBM is a formalization of the life cycle of an object in regard to the following parts:

- States
- Events
- Transitions
- Actions

The state of OBM corresponds to the state of an objects life cycle. An *event* is an incident or action that causes a progression to another state or to the same state. The *transition* is the new state of an object if a particular event occurs while the object is in a particular state. The *action* is the function that is performed immediately when entering a new state. Each state can have only one action, but that action can consist of many processes.

Figure 19−10 shows a sample OBM. The boxes represent the states, and the lines represent the events. The event causes the transition to the new state.

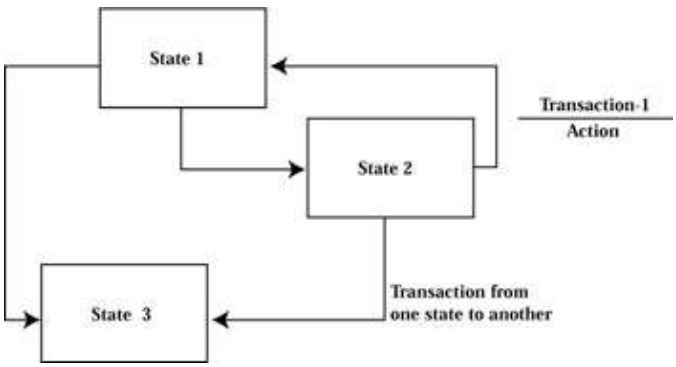

Figure 19−10: OBM sample

An alternate form of state model representation is the state transition table (STT). In an STT the rows represent states, the columns represent events, and the cells represent the effect of each event in each state. When a particular event cannot happen for a particular state of the object, then the cell entry cannot happen.

The event may be prevented from happening for many reasons, such as a physical impossibility, definition of the object, or constraints. If a particular event can happen for a specific state of the object but the object does not respond to the event, then an event−ignored entry is made in that cell. If the event is ignored, then the object stays in the same state. The advantages of an STT are that it can be extended by adding an additional column for action and is associated with entering each state. The information contained in the STT or state transition diagram (STD) is the same, but the development of the STT from the STD will catch one of the most common errors in OBM development.

OBM is built for each dynamic object in the OIM. The following are guidelines for construction of an OBM:

- Take one instance of the model and analyze and record the life cycle of that instance.
- Write down the various states for that instance.
- Find the states that consider all relationships of the object.
- Build an STT.
- Check the STT for new or additional states.
- Define the action that will be executed upon entry into a state.
- Identify nonfinal states.
- Identify events that the object must generate so it can exit nonfinal states.
- Expand actions that generate these events.
- Add the new events to the STT.
- Complete the STT.
- Complete the STD.

Object Process Model

The OPM makes use of data flow diagrams and develops the required processes that drive the objects through their event chains. Only a few concerns with objects are discussed in this section.

The following is a list of processes for the development of a data flow diagram of a single object:

- Develop an OBM that formalizes the behavior of an object over time.
- Analyze the action that is performed when the state is entered.
- Break the action down into a sequence of processes.
- Depict each process as a data flow diagram.
- Place each action data flow diagram on a separate page.

The data of an object are represented as a data store in the data flow diagram. The data store will be recorded in the ODD. The data store for the object contains all of the attributes of the object. The instances of the object are created and stored in the data store. The data store holds all data and all attributes of all instances of the object. The data store is shared by all action data flow diagrams. The data flow diagram involves data stores, which are made up of attributes of that data. These processes are discrete.

Structured Design Method

The structured design method maps the flow of data from its problem domain into its software structure (Figure 19−11). The steps of structured design involve characterization of the data flow through graphic representation, identification of the various transform elements, assembly of these elements in a hierarchical program structure, and refinement and optimization of the elements. The structured design uses the following terms:

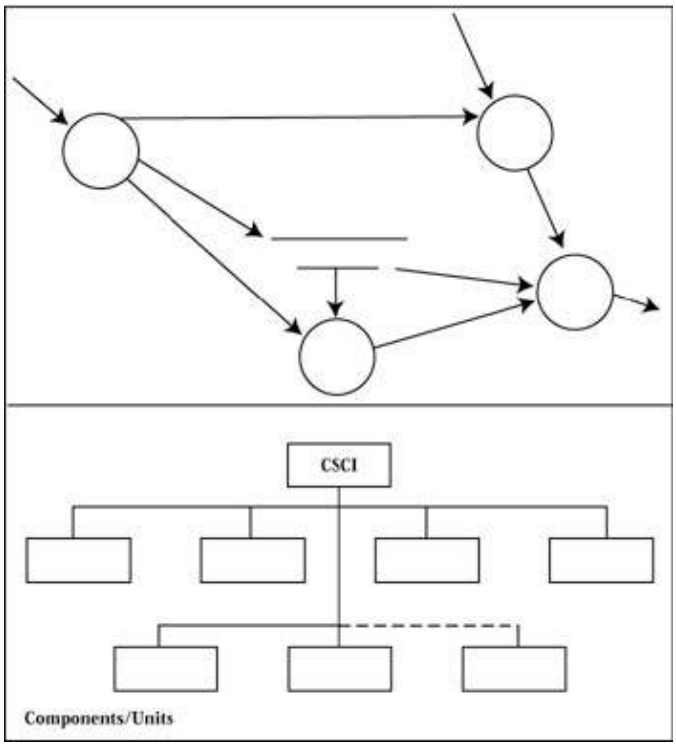

Figure 19−11: Software−structured design

- ***Efferent.*** This is the flow of information out of a software system, such as the output flow. Efferent modules take information from superordinates, possibly perform some data transformation, and then pass the information onto another subordinate module. The primary purpose of afferent and efferent modules is to pass information into and out of the software system. As afferent modules pass information into the software system, the information becomes less recognizable as input data, which is more abstract. As the efferent flow is traced backward into the software system, it becomes less recognizable as output data. Points of higher abstraction of input and output data are further removed from the physical inputs and outputs. These points still constitute as inputs to and outputs from the software system. The points of the highest abstraction for the major input and output streams define the afferent, efferent, and central transformation portions of a software hierarchy. Structured design does not provide precise guidelines for determining the points of highest abstraction. Designer intuition and experience are required to make these determinations.
- ***Central transforms.*** These are modules whose primary function is to transform information from one form into another and perform the primary work of the software system. Transform flows move data from the invoking module to the subordinate module, if the subordinate module has performed some transformation, and then back to the invoking module.
- ***Structure chart.*** Developers use the data flow to develop a structure chart by performing top−down decomposition of the central transforms. The structure charts present the partitioning, hierarchy and organization, and communication among software modules. A module is a set of one or more program statements that is invoked by other parts of the software system. Developers use the structure chart to

develop the data structure. They use the results to reinterpret the functional requirements as described in the software requirements specifications. A module is equivalent to a software item, unit, or component.

- ***Fan−out.*** This is the number of subordinate modules that a module possesses. A high fan−out can lead to a flat software system that has no intermediate levels in its structure
- ***Fan−in.*** This is the number of invoking modules that a module possesses. A fan−in means that duplicated code was avoided.
- ***Coupling.*** This is a measurement of relationships among modules and is used to evaluate various program organizations. It is a measure of the strength of interconnection when the strength of coupling and intermodule dependence is directly related. Strong coupling complicates a system because a module is hard to understand, change, or correct by itself if it is highly interrelated with other modules. Developers can reduce complexity by designing a module with the weakest coupling possible between modules.
- ***Cohesion.*** This is a measurement of the strength of association of elements within a module. Modules with a high degree of cohesiveness are understandable and excellent candidates for reuse.
- ***Heuristic design.*** In addition to coupling and cohesion, structured design discusses a heuristic design that is used to organize modules and decisions. Developers use it as a check or indicator by which a structure can be examined for potential improvement.
- ***Module size.*** The size of the module should be small enough to be understandable.
- ***Span of control.*** This is the number of modules immediately subordinate to a given module and should be between 2 and 9. Spans of control beyond this range usually indicate that the modules function is too complex.

When assessing the wide popularity of structured design, L. Peters (1981) stated the following:

Structured design has gained wide popularity for two primary reasons. One is that it allows the software designer to express his perception of the design problem in terms he can identify with: data flows and transformations. The notation with which he expresses these flows is simple, easy to use, and understandable by management, customer, and implementers.

The other primary reason for this methods popularity is that it provides the designer with a means of evaluating his (and others) design. This serves as a benchmark against which to measure his success or progress. The method is unique in this regard. If the design evaluation concepts consisted of nothing more than coupling and cohesion, structured design would still be a significant contribution to the software fields.

Software Design Description

The software design description (SDD) contains a detailed description of a software item. With the software architecture, it includes all the detail that is needed to implement the software. The SDD is a representation of a software system that is used as a medium for communication of software design information. The SDD contains internal interfaces among the software components and units, networking design, and database design. The SDD includes the following:

- Software design architecture
- Description of how the software item satisfies the software requirements, including algorithms and data structure
- Software item input and output description
- Software item relationships
- Concept of execution, including data flow and control flow
- Requirements traceability

    - ♦ Software component−level requirements traceability
    - ♦ Software unit−level requirements traceability
- Rationale for software design
- Reuse element identification
- Definition of types of errors and their handling
- ***Design entities and attributes.*** A design entity is an element (component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced. Design entities result from a decomposition of the software requirement. A design entity attribute is a named characteristic or property of a design entity. It provides a statement of fact about the entity. An incorrect specification of the attribute value can result in a fault in the software implementation.
- ***Identification.*** Unique name of the design entity. Two entities should not have the same name.
- ***Type.*** Description of the kind of entity, such as subprogram, module, procedure, process, or data store. Alternatively, design entities may be grouped into major classes to assist in locating an entity dealing with a particular type of information.
- ***Purpose.*** Description of why the entity exists (rationale for the creation of the entity). The purpose attribute should also describe special requirements that must be met by the entity that are not included in the software requirement specification.
- ***Function.*** A statement of what the entity does. The function attribute states the transformation applied by the entity to input to produce the desired output. In the case of a data entity, this attribute states the type of information stored or transmitted by the entity.
- ***Subordinates.*** Identification of all entities composing this entity. This information is used to trace requirements to design entities and to identify parent−child structural relationships through a software requirement decomposition.
- ***Dependencies.*** A description of the relationships of this entity with other entities. These relationships are graphically depicted by structure charts, data flow diagrams, and transaction diagrams. This attribute should describe the nature of each interaction, including such characteristics as timing and conditions for interaction. The interactions involve the initiation, order of execution, data sharing, creation, duplication, usage, storage, or destruction of entities.
- ***Interface.*** A description of how other entities interact with this entity. The interface attribute describes the methods of interaction and the rules governing those interactions. The methods of interaction include the mechanisms for invocation or interruption of the entity, communication through parameters, common data areas or messages, and direct access to internal data. The rules governing the interaction include the communications protocol, data format, acceptable values, and the meaning of each value. The interface attribute provides 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. It also includes inputs, screen formats, and a complete description of the interactive language.
- ***Resources.*** A description of the elements used by the entities that are external to the design. The interaction rules and methods for using the resource are specified by this attribute. The attribute provides information about items such as physical devices (printers, disc−partitions, memory banks), software services (mathematics libraries, operating system services), and processing resources (central processing unit cycles, memory allocation, and buffers). The resources attribute describes 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 also includes the identification of potential race and deadlock conditions as well as resource management facilities.
- ***Processing.*** A description of the rules that the entity uses to achieve its function. The processing attribute describes the algorithm that the entity uses to perform a specific task and includes contingencies. The description includes 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 describes the action to be taken in the case of overflow conditions or in the case of a validation check failure.

- ***Data.*** A description of data elements internal to the entity. The data attribute describes the method of representation, initial values, use, semantics, format, and acceptable values of internal data. The description of data is in the form of a data dictionary that describes the content, structure, and use of all data elements. It includes data specifications such as formats, number of elements, and initial values. It also includes the file structures, arrays, stacks, queues, and memory partitions. The description includes static versus dynamic data elements; whether it is to be shared by transactions, used as a control parameter, or used as a value; and loop iteration count, pointer, or link field. The data information includes a description of data validation needed for the process.