The Object-Oriented Software Development Method: A Practical Approach to Object-Oriented Development

by Edward Colbert
Absolute Software Co., Inc.

The Object-Oriented Software Development Method (OOSD) includes object-oriented requirements analysis, as well as object-oriented design. OOSD is a practical method of developing a software system which focusses on the objects of a problem throughout development. OOSD's focus on objects early in the development, with attention to generating a useful model, creates a picture of the system that is modifiable, reusable, reliable, and understandable — the last perhaps most important because the picture of a software system created by a development method must be an effective device for communication among developers, customers, management, and quality-assurance personnel.

Most object-oriented methods competing for the attention of the software developer actually apply traditional Structured Analysis (function-based), or variations of Structured Analysis, to requirements activity, and work through a transition process to an object-oriented design [1,2,7,10,11]. In these methods the developer begins with functionally-based requirements analysis, and only reaches an object-oriented design by the intermediary step of converting a traditional, functionally-decomposed data flow diagram (DFD) to an object-oriented DFD (or equivalent). In this conversion process, objects are identified through a set of heuristics which group "transformations" in the DFD generated during requirements analysis. These methods carry a number of interesting but unfortunate burdens. Lower-level objects, which directly relate to real-world objects, are easily identified, but higher-level objects are generally more arbitrary, so that developers do not consistently identify a hierarchy of objects which achieves significant improvement in software engineering goals (e.g., reliability, maintainability, reusability). The heuristics for identifying objects usually relate the DFD transforms to the object that controls execution of an operation, rather than the object which "owns" the operation. These methods generally ignore the need to convert behavior descriptions of the DFD transforms into behavior descriptions of the objects. Finally, the use of Structured Analysis in an otherwise object-oriented approach complicates the tracing of requirements by forcing the developer to look first to DFD transforms and their behavior descriptions, and then to the objects.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

WHAT IS AN OBJECT?

Critical to the effectiveness of an object-oriented method is the definition of *what is an object* and *what are its characteristics*. The answer to the first question helps find the objects that need to be represented in a model of the problem. The answer to the second question tells what needs to be represented about the objects.

An *object* is a visible or tangible thing of relatively stable form; a thing that may be apprehended intellectually; a thing to which thought or action is directed [9]. It is a *unit of structural and behavioral modularity* [4] which has properties. Each object can be characterized by the set of operations that can be performed on it and by the set of operations that it can perform on other objects (either of these two sets may be empty), and by the set of states which it goes through during its lifetime [2]. Each object encloses components, i.e. objects of which it has been constructed, and the operations that can be performed on it, which it makes available for other objects to perform. When these characterizations are accurate, we recognize an object: a *self-contained thing* which exhibits behavior.

OOSD considers an object *active* if it displays independent motive power; if it does not, it is considered *passive*. An *active* object need not exercise its motive power (for example, a human being sometimes acts in response to requests or commands), but a *passive* object acts only under the motivation of an active object.

A *class* defines a number of objects that share the same set of states, the same set of operations that can be performed on them, the same set of operations that they can perform on other objects [2], and the same class of component objects. By identifying a class, we can avoid redundancy in defining the objects that are members of that class.

In developing systems, we are constantly concerned with *data* about objects; a datum is an individual fact, statistic, or

© 1989 ACM 0-89791-329-9/89/0010-0400 $1.50

piece of information derived from history, observation, calculation, or experiment [9]. Data describe an object's attributes or state (or both).

OOSD seeks to identify the objects in a problem, to understand the structural and behavioral modularity and properties of each object, and to recognize objects which are members of a common class and so share modularity and properities, so as to create a single consistent abstract model based on the elements of the problem. In requirements analysis, this model identifies the "what": the required objects, classes, functions, behavior, and attributes of the problem. In design, this model determines the "how": it is refined into an architecture for software components with a smooth transition to code. The model is developed and viewed through graphic and textual representations which provide ready communication.

OBJECT-ORIENTED REQUIREMENTS ANALYSIS

In OOSD requirements analysis, four activities are performed to create the model of the application from the problem statement. These activities (illustrated in Figure 1) produce graphic and textual representations of the model.

1) Object-Interaction Specification
2) Object-Class Specification
3) Behavior Specification
4) Attribute Specification

The activities can be performed in almost any order once an initial Object-Interaction Specification has been performed. An iterative approach is recommended, since the performance of each activity generally suggests refinements in products of the other activities.

Object-Interaction Specification

The Object-Interaction Specification activity identifies the required objects, their interactions, and the required hierarchical relationship of objects. Two graphic representations are used in this activity, Object-Interaction Diagrams (OID) and Object-Hierarchy Diagrams (OHD). The Object-Interaction Diagrams describe a set of objects and the interactions between the objects (see Figures 2a-d). The Object-Interaction Diagram is the fundamental representation of the model of the application.

In an OID, objects are symbolized by three different types of rectangles. rounded for active objects (e.g., *Monitor*, *Sensor*, *User* in Figure 2b), open for passive objects (e.g., *Alarm*, *Data_Log* in Figure 2b), and ordinary rectangles for external objects (e.g., *User*, *Physical_Sensor* in Figure 2a). *External* objects are objects outside the scope of the system to be implemented; for active and passive objects, see "What is an Object?" above.

An interaction involves an operation and (optional) information flows. Operations are symbolized in an OID by arrow-shaped arcs connecting the objects which interact. In OOSD notation, an operation is **initiated by the object** shown at the tail of the arrow in the interaction; **performed by** the object shown at the head of the arrow; the operation is called an **operation on** the object that performs the operation. Information flows in an interaction can be data flows, object flows, or error flows, symbolized by the three types of couples illustrated in Figure 3 and shown in other figures. For example, in Figure 2a the *Execute* interaction between the *User* object and the *Temperature_Monitor_System* object consists of the *Execute* operation; 4 data flows, namely *Command*, *Sensor_Name*, *Limits*, and *Sensor_Log*; no object flows; and 2 error flows, namely *Invalid_Command* and *Bad_Data*. The *Execute* operation is performed by the *Temperature_Monitor_System* object and initiated by the *User* object.

The decomposition of objects (active and passive) and their interactions in the model is represented by hierarchical Object-Interaction Diagrams comparable to the hierarchy of DFDs used in Structured Analysis. The top OID is a **Context Diagram.** It describes the system object in the context of those objects outside the system which the system object interacts with (see Figure 2a).[1] Additional OIDs are created to describe the internal structure of an object, and called internal or "exploded" views. Each internal-view diagram describes:

1) the component objects of one object in a higher diagram;
2) the interactions among the component objects;
3) the interactions between the components and their parent object;
4) the relation between interactions performed *on* the parent object and interactions performed *on* the component objects;
5) the relation between interactions initiated *by* the parent object and interactions initiated *by* the component objects.

For example, a diagram is created that describes the internal structure of the system — note that the system itself is an object. The internal view of the system object describes component objects of the system object and how the system and component objects interact (see Figure 2b).

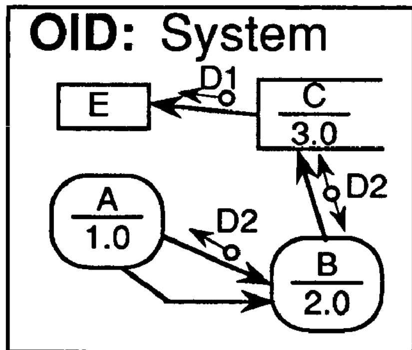
An object which has an internal view is distinguished from one that does not by highlighting its graphic representation with bold lines (see Figures 2a-b). Each internal-view diagram illustrates the fact that the components are part of a higher-level object by placing the components inside a representation of the higher-level object (see Figures 2c-d). The objects that the parent object interacts with are included in a decomposition diagram of the parent object as off-page connectors with the interactions drawn between the connectors and the parent object (e.g., in Figure 2b, *User*, *Physical_Sensor*, *Hardware_Timer*, *Sensor_Limit_Light*, and *Sensor_Fault_Light*).[2]

There are four possible relations between interactions on or initiated by the parent object with interactions on or initiated by the component objects of a parent.

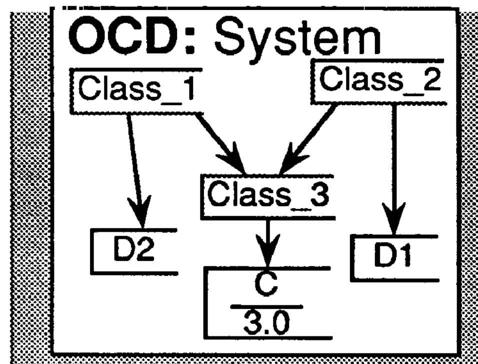
[1] Interactions between the system object and external objects are interface interactions, symbolized by a dashed line.

[2] There are consistency rules for interactions similar to the rules for consistency of data flows in DFDs.

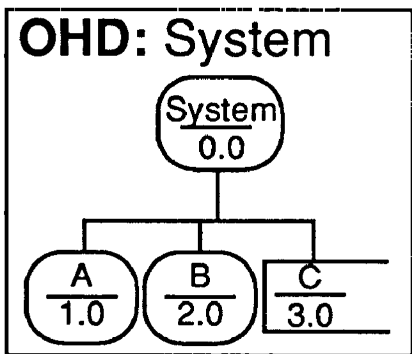
Object-Interaction Specification

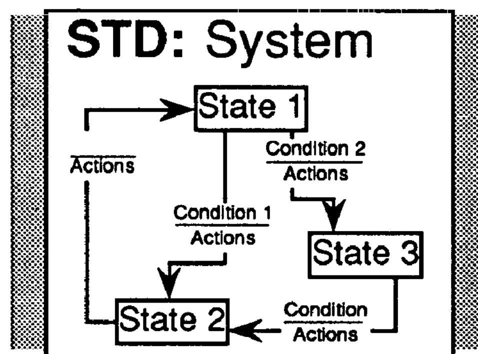


Object/Class Specification



Behavior Specification





Attribute Specification

AT: System

	Scale	Test	Worst	Planned	Record	Past
Workability						
Reusability						
Portability						
Performance						
Useability						

Summation of Results

SRD

	Subsystem	Description	Entity	Category	Object Class	Notes
A						
B						
C						
D1						
D2						

Figure 1: Specification Activities and Representations

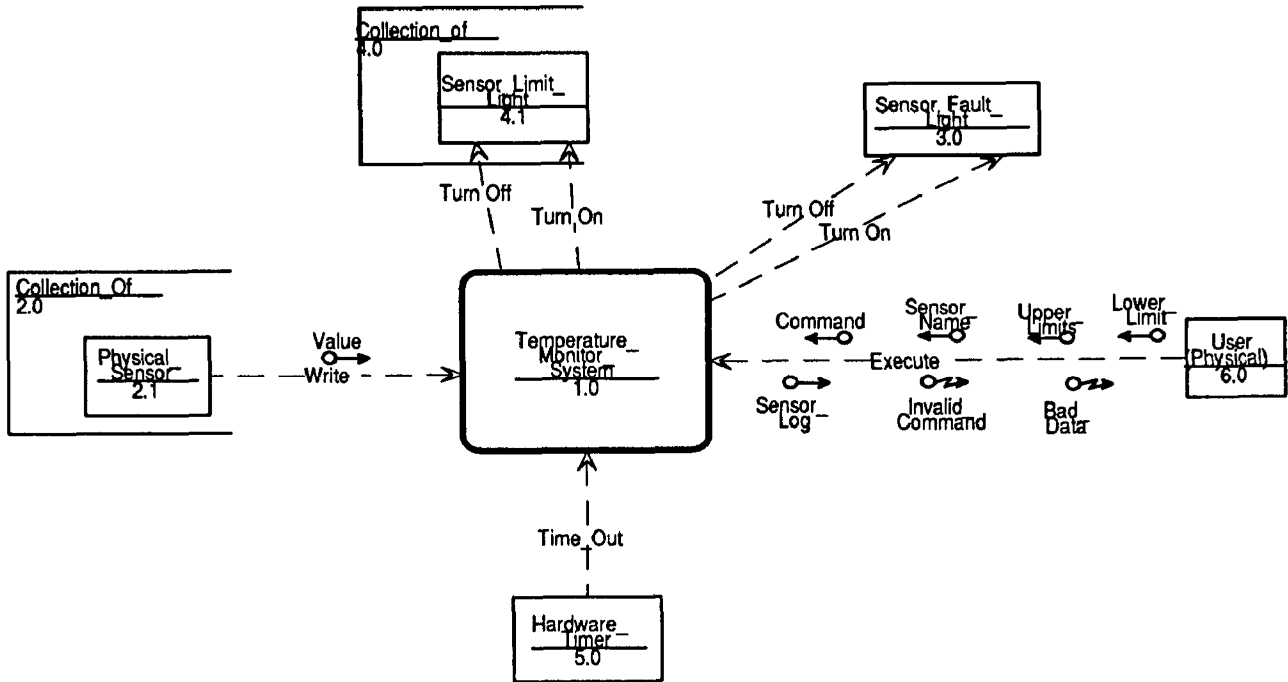


Figure 2a: Context Object-Interaction Diagram for the Temperature Monitor System [3].

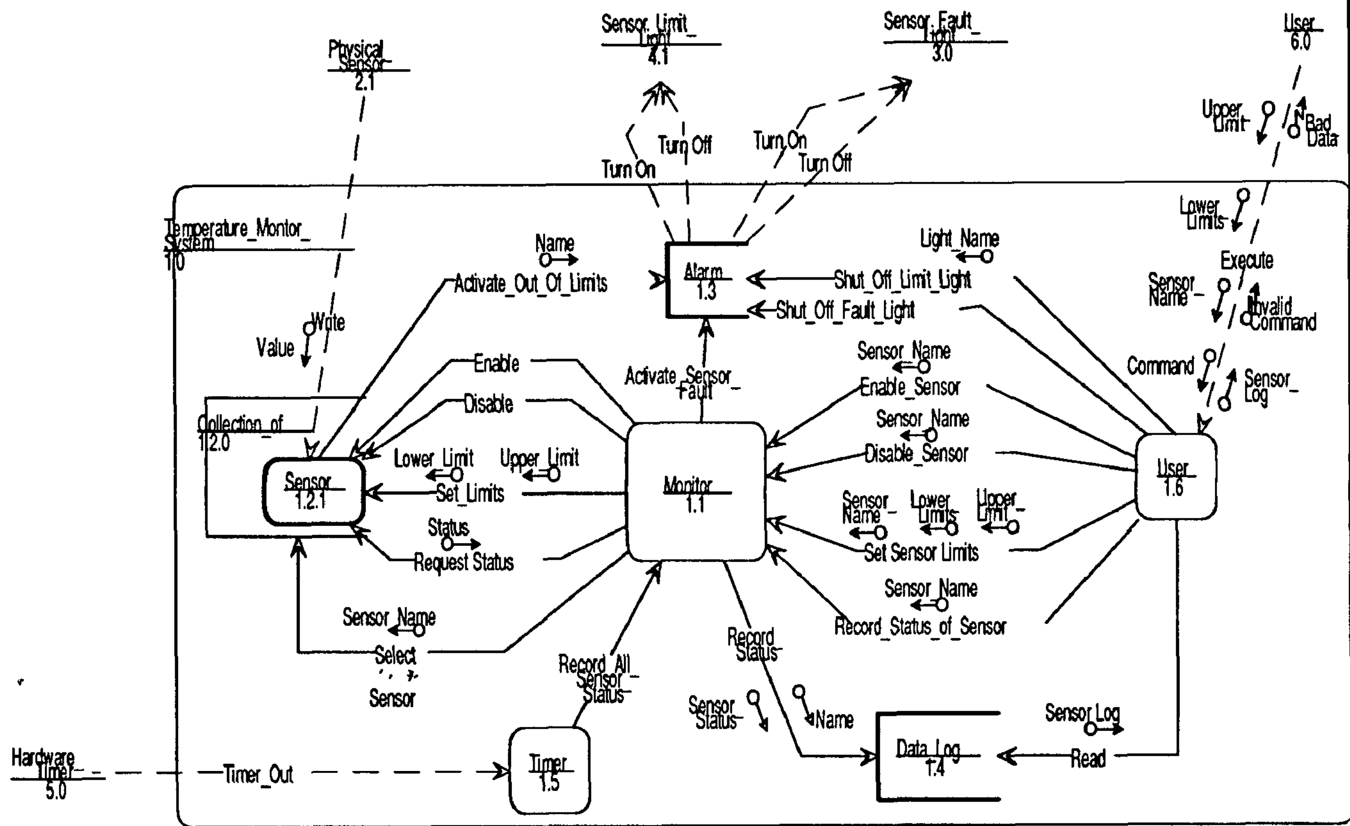


Figure 2b: Object-Interaction Diagram 1.0: Temperature_Monitor_System [3].

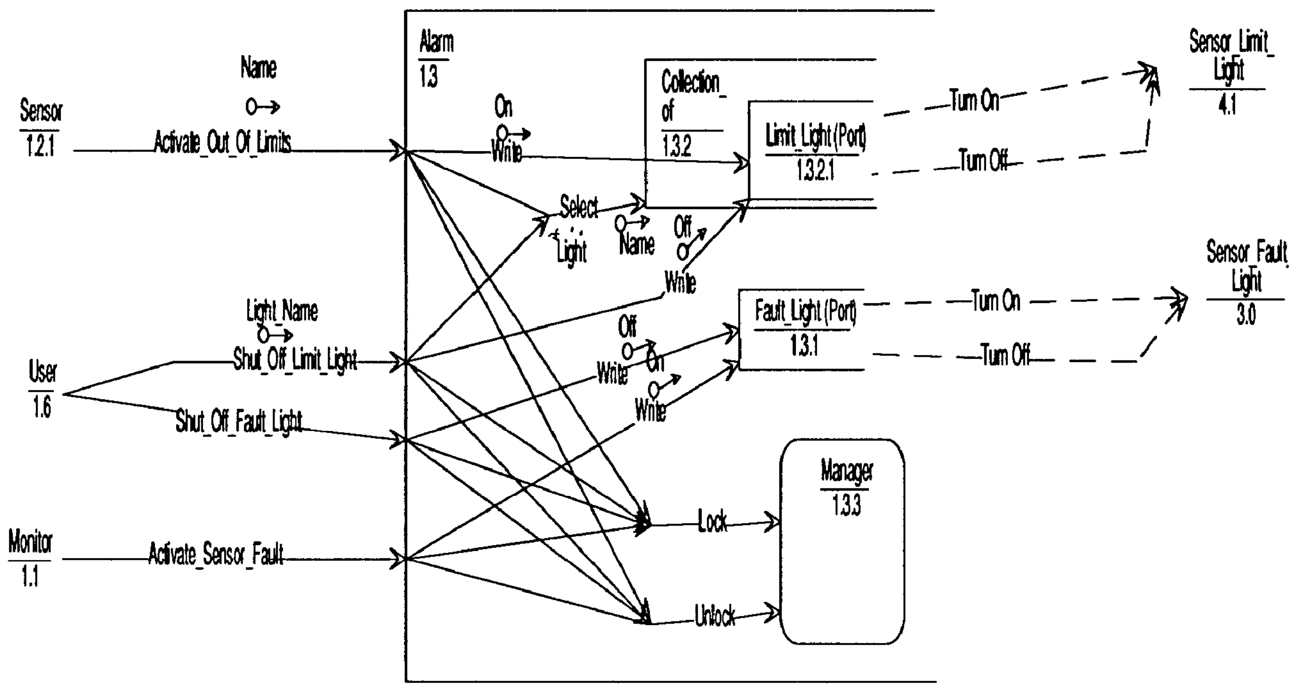


Figure 2c: Object-Interaction Diagram 1.3: Alarm Object.

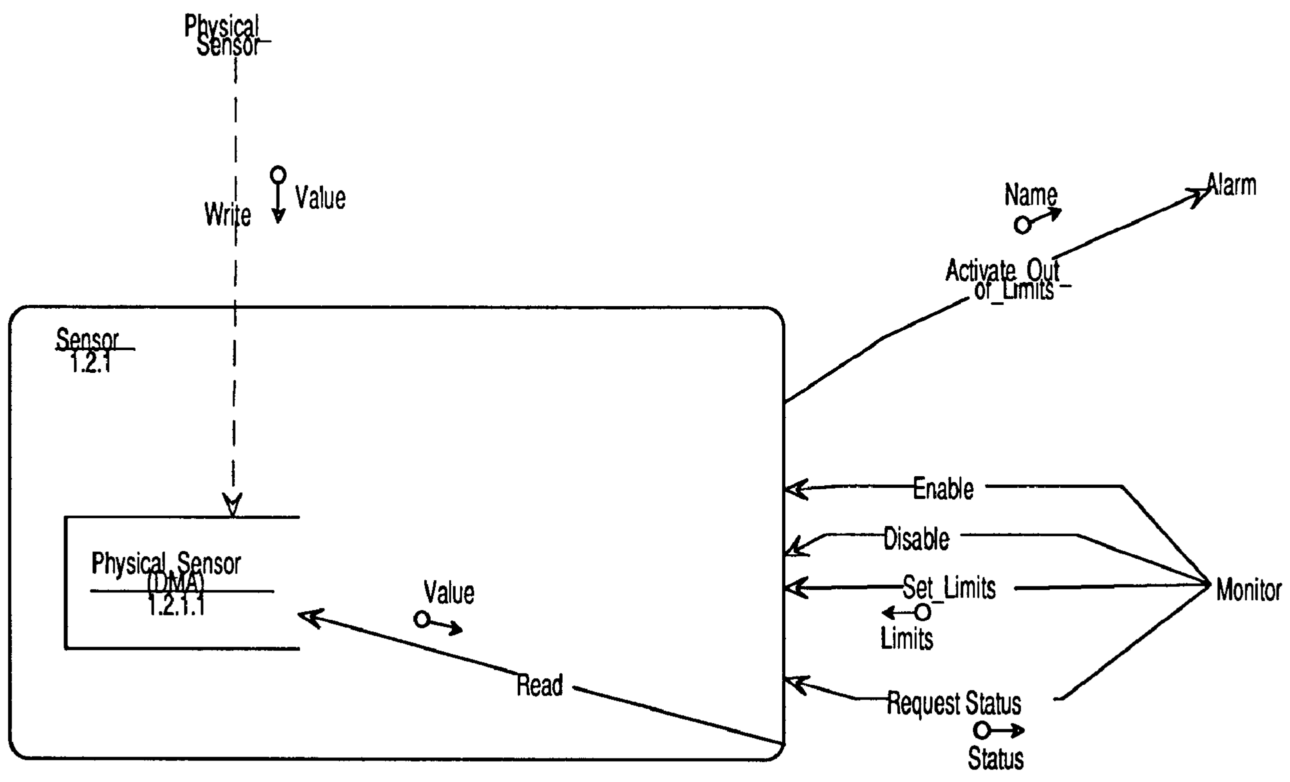


Figure 2d: Context Object-Interaction Diagram: Sensor Object

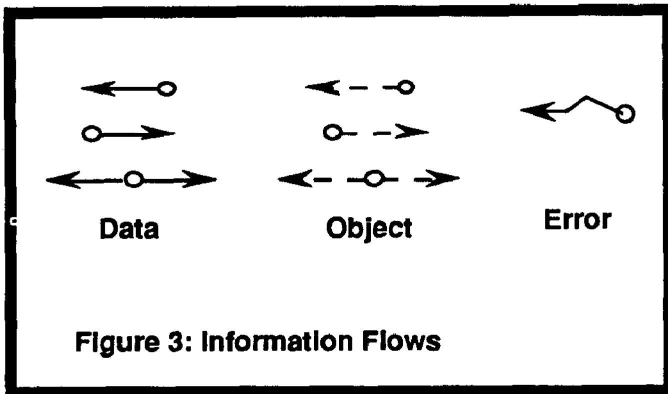


Figure 3: Information Flows

1) An interaction on a parent object is decomposed into interactions on component objects of the parent. For example, in Figure 2c, the *Activate_Out_of_Limits* operation on the *Alarm* object is decomposed into two operations on component objects: the *Select* operation on the *Collection_of (Limit_Light)* and the *Write* operation on a *Limit_Light* object. (Note: decomposition of information flows is also possible; however, it does not occur in this example at this level of detail).

2) An interaction initiated by a parent object is only conceptual and is actually initiated by one or more component objects of the parent object. For example, in Figure 2a the *Temperature_Monitor_System* object is pictured conceptually as initiating the operations *Turn_On* and *Turn_Off* on the external objects *Sensor_Fault_Light* and *Sensor_Limit_Light*. Figures 2b–c show that these operations are actually initiated by the *Sensor_Fault_Light (Port)* and *Sensor_Limit_Light (Port)* objects, respectively[3].

3) An interface interaction on a parent object is only conceptual and is actually an interaction directly on a component object of the parent object. For example, Figures 2a,b,d show that the *Write* interface interaction is conceptually on *Temperature_Monitor_System*, but is actually on *Physical_Sensor (DMA)*.

4) An interaction on a component object is initiated by the component's *active* parent object, independent of the initiation of any interaction on the parent (*passive* parent objects cannot initiate an operation on a component independent of the initiation of any interaction on the parent, since passive objects lack "volition"). For example, Figure 2d shows that the *Sensor* object initiates the *Read* operation on the *Physical_Sensor (DMA)* "at will" (i.e., under some condition other than response to an interaction on *Sensor*).

The Object–Hierarchy Diagram summarizes, for communication purposes, the hierarchical relationship established in Object–Interaction Diagrams. For example, Figure 4 is the OHD for the set of OIDs shown in Figures 2a–d. OHDs can be generated automatically from OIDs.

[3] This representation is used any time a component object initiates an operation of an object defined at the same (or higher) level of the parent object.

Object–Interaction Specification Approach

Three approaches for the development of the Object–Interaction Specification seem especially useful. One, an adaptation of the "Outside–In" approach as described for DFDs by Nielson & Shumate [7], starts with the context diagram, decomposes the system by creating an interface object for each internal, and works inward. A second approach, described by Ray Buhr as "Behavioral Partitioning" [4], analyzes the concurrent behavior in a problem and uses that information to determine the objects. These two approaches are useful when only the context is well known. A third approach, described by Ken Orr [8], can be characterized as "Middle–Up–Down", identifying the components of the system object, then working up to define the context, and finally working down again to refine the system and its components. This approach is useful when reworking an existing system (computer or manual).

Whichever approach is applied, the goal should be the identification of those objects needed to model "what" the system does. The system object needs to be decomposed *during requirements analysis* only to a level that accounts for all interface objects to external objects, and correctly describes the interactions.

Object–Class Specification

The Object–Class Specification activity identifies the classes of objects, and the relationships between the classes, using Object–Class Diagrams (see Figure 5). The Object–Class Diagrams describe the class of each object and information flow in the Object–Interaction Diagrams of a system, and describe which classes are related to other classes in the system. In identifying the class of an object, the structural and behavioral modularity and properties of the object are established for the class.

For example, Figure 5 is a consolidated form of the Object–Class Diagrams for the *Temperature_Monitor_System* described in Figures 2a–d. It conveys the following information:

1) all of the classes identified during the requirements analysis (*Sensor_Type, Status_Type, Temperature_Type*, etc);

2) which objects and information flows in the Object–Interaction Diagrams of the system are members of which class (e.g. the *Sensor* object and *Sensor* object flow of *Collection_of.Select* operation are both members of the class *Sensor_Type*; similarly the *Physical_Sensor (DMA)* object and the *Value* data flow of the *Physical_Sensor (DMA).Read* operation are members of the class *Temperature_Type*);

3) the relation of some of the classes (e.g., the *Status_Type* has two components of *Temperature_Type* and one component of *State_Type*); and

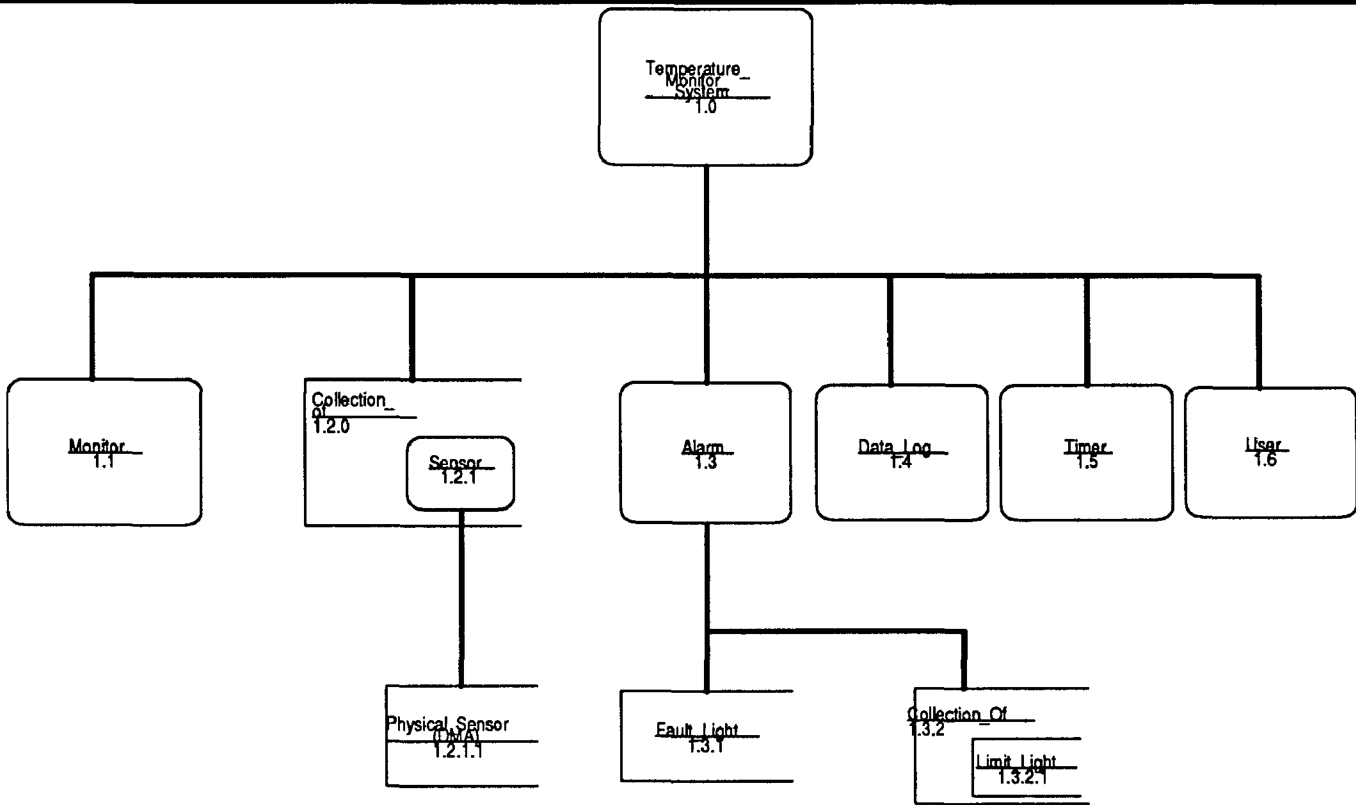


Figure 4: Object-Hierarchy Diagram for the Temperature Monitor System [3].

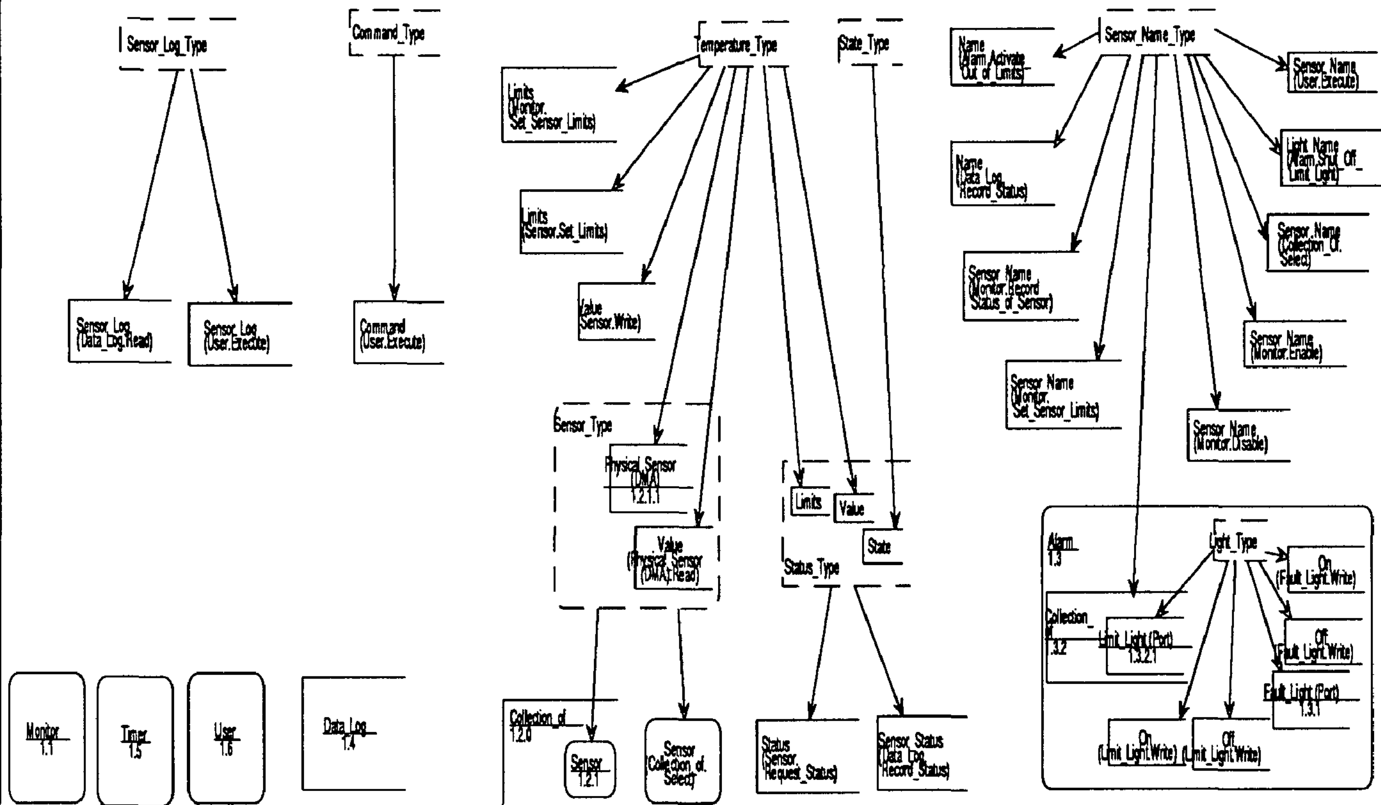


Figure 5: Object-Class Diagram for the Temperature Monitor System [3].

4) the structural and behavioral modularity and properties of the class (e.g. *Sensor_Type* is a class of active objects; all other types shown are classes of passive objects). Note: the operations associated with each class are described textually.

During requirements analysis, only those relations which are actually required are described. For example, in the *Temperature_Monitor_System,* there must be some relation between the *Data_Log* object and the *Status_Type* and *Name_Type* class; however, the nature of the relation is a design decision.

Behavior Specification

The Object–Interaction Diagram gives a **static** view of the behavior of each object in the system. Object–Interaction Diagrams show *which objects interact,* but not *under what conditions.* For example, Figure 2b does not show whether initiation of the *Activate_Sensor_Fault* operation on the *Alarm* object causes initiation of the *Turn_On* operation of the *Sensor_Fault_Light* object.

The Behavior Specification activity identifies the **dynamic** behavior of the system and of each object in the system. Mealy State Transition Diagrams (see Figure 6) are used to represent behavior; however, other graphic representations could be used.

The Behavior Specification activity has three major steps.

1) Describing the behavior of the system defined in the problem statement and analyzing the description for correctness and completeness.

2) For each object in the model, defining both external behavior and internal behavior. The external behavior of an object relates the operations performed on it to the operations it performs on other objects (see Figures 7a,c,d). The internal behavior of an object relates the operations performed on it to the operations it performs on its component objects (see Figure 7b).

3) Demonstrating that the behavior of all objects in the model is correct. This demonstration is accomplished when the correct behavior has been described for each object, and each object is correctly implemented by its components. The process has four steps, which may be automated by support tools.

   a) Demonstrating that the external view of the behavior of an object correctly relates to the external view of the object in the Object–Interaction Diagram.

   b) Demonstrating that the internal view of the behavior of an object correctly relates to the internal view of the object in the Object–Interaction Diagram.

   c) Demonstrating that the external view of the behavior of an object is correctly implemented by the combination of the internal view of the behavior of that object and the external views of its component objects (e.g., compare the behavior described in Figures 7b–c with the behavior in Figure 7a).

   d) Demonstrating that the required behavior as defined by the problem description is correctly implemented by the internal view of the system object and the external views of its components.

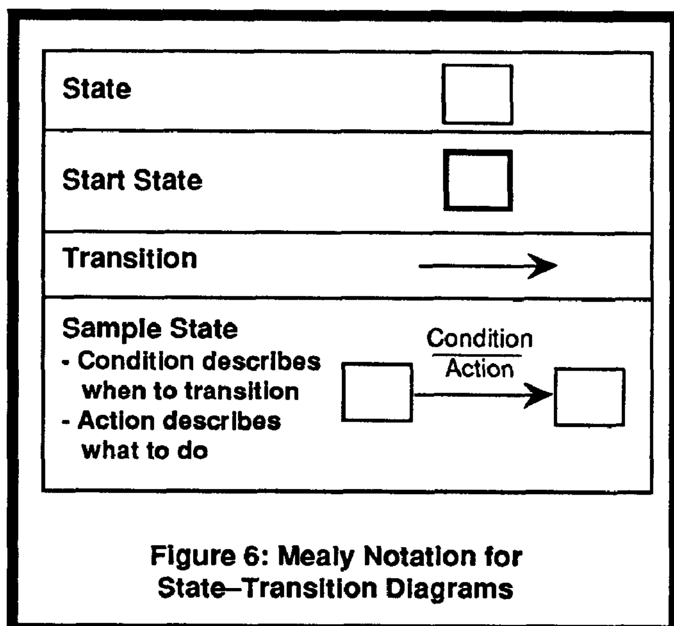


Figure 6: Mealy Notation for State–Transition Diagrams

Attribute Specification

The Attribute Specification activity identifies the quantitative and qualitative measures and resources for the each object in the system. Two textual representations are used: the Attribute Specification Form and the Attribute Summary Table [6]. The Attribute Specification Form (see Figure 8) states the unit of measurement for each quality or resource of the system (or an object in the system), and the minimum and desired measures to be met. The Attribute Summary Table summarizes the description of all attributes of the system (or an object in the system), and helps analyze trade–offs of attributes, risks of failing to achieve desirable or necessary combinations, and alternate designs.

Object–Oriented Requirements Analysis Results

OOSD requirements analysis results in a complete model of the problem, describing "what" is needed by means of a hierarchical set of objects whose interactions, behavior, and attributes represent the problem correctly. The model is shown by the graphic and textual views described in the preceding sections. In addition, a System Requirements Dictionary is created to summarize the results and to cross–reference the information in the various representations.

OBJECT–ORIENTED DESIGN

The OOSD approach to design refines the model produced in the requirements analysis phase into a software architecture, and defines a language–specific representation of that architecture.

Figure 7a: State Transition Diagram 1.3, External View of Alarm Object

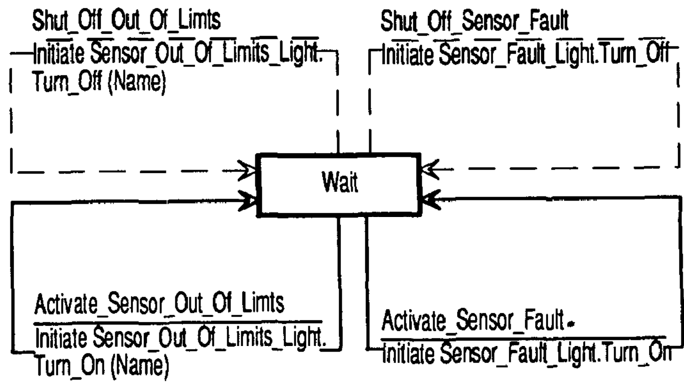
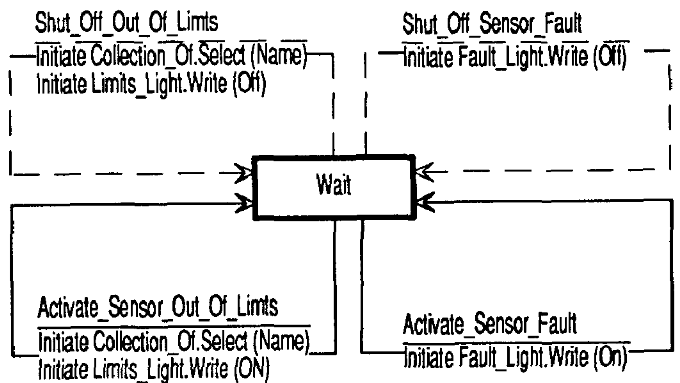


Figure 7b: State Transition Diagram 1.3, Internal View of Alarm Object



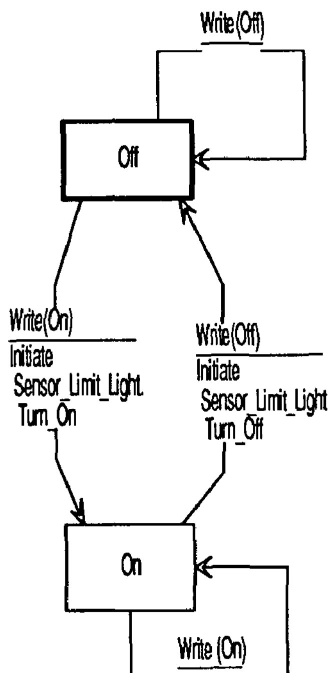


Figure 7c: State Transition Diagram 1.3.2.1, External View of Sensor_Fault_Light (Port) Object

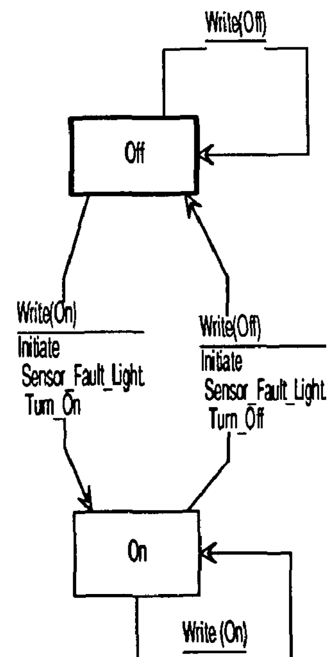


Figure 7d: State Transition Diagram 1.3.1, External View of Sensor_Fault_Light (Port) Object

Figure 8: Attribute Specification Example

Sensor .Responsiveness.Maximum_Processing_Time.Repeated_Interaction:

Description=	**The maximum time which a sensor object can take in performing any action (i.e. an interaction or the reading and testing of the physical sensor value (including the recording of out–limits conditions)**	
Scale=	**Seconds**	
Test=	**For each operation of the sensor, the operation will be initiated repeatedly for 1 hour period at a fixed interval, starting at the plan level, increasing by 1 second units, until each interaction during the hour succeeds. The interval at which all interactions succeed, defines the value for this operation. The maximum time interval for all operations will define the value for this attribute.**	
Worst=	**5 seconds**	**(average of times for individual interaction)**
	5 seconds	**(each interaction)**
Plan=	**3 seconds**	**(average of times for individual interaction)**
	1 second	**(each interaction)**
Record=		
Past=		
Note=	**The plan level of average interaction is design as a 40% safty margin. The worst case is defined by contract.**	

Preliminary Design

The preliminary design creates a language–independent description of the software solution to the problem. The goal of preliminary design is to specify the software architecture for the system. To accomplish this goal, the model of the application which was developed during requirements analysis is made the top–level architecture. This architecture is refined taking into consideration "how" the model will be implemented (see, for example, the elaboration of Figure 2d shown in Figure 9). During preliminary design the activities of requirements analysis take new meaning as they are used again to elaborate the architecture. In this way, the problem model is refined into a solution model. The products of this activity appear as elaborated versions of the now familiar Object–Interaction Diagrams, Object–Hierarchy Diagrams, Object–Class Diagrams, State–Transition Diagrams, and Attribute Specification Forms and Tables.

Detailed Design

The detailed design creates an implementation–specific representation of the software architecture coupled to the implementation language, detailed enough to produce code. The goal of detailed design is to specify an implementation of the software architecture produced in preliminary design. In OOSD, four activities are performed during detailed design parallel to those done during requirements analysis.

1) Language–Specific Software Architecture Specification
2) Object–Class Specification
3) Behavior Specification
4) Attribute Specification

These activities (illustrated in Figure 10) modify the graphic and textual representations of the model already produced, and produce new representations. Once an initial Software Architecture Specification has been performed, the activities can again be performed in almost any order; since the performance of each activity generally suggests refinements in products of the other activities, an iterative approach is recommended.

Language–Specific Software Architecture Specification

The Software Architecture Specification activity describes, in a language–specific notation, the software components and structure defined by the architecture developed during preliminary design. For each object and class defined in Object–Interaction Diagrams and Object–Class Diagrams, decisions are now made on how to represent these objects and classes in the implementation language.

Using the Ada language as an example, in Ada design a modified form of the notation developed by Ray Buhr [5] is used for graphic representation, and an Ada Program Design Language (Ada PDL) is used for textual representation. There

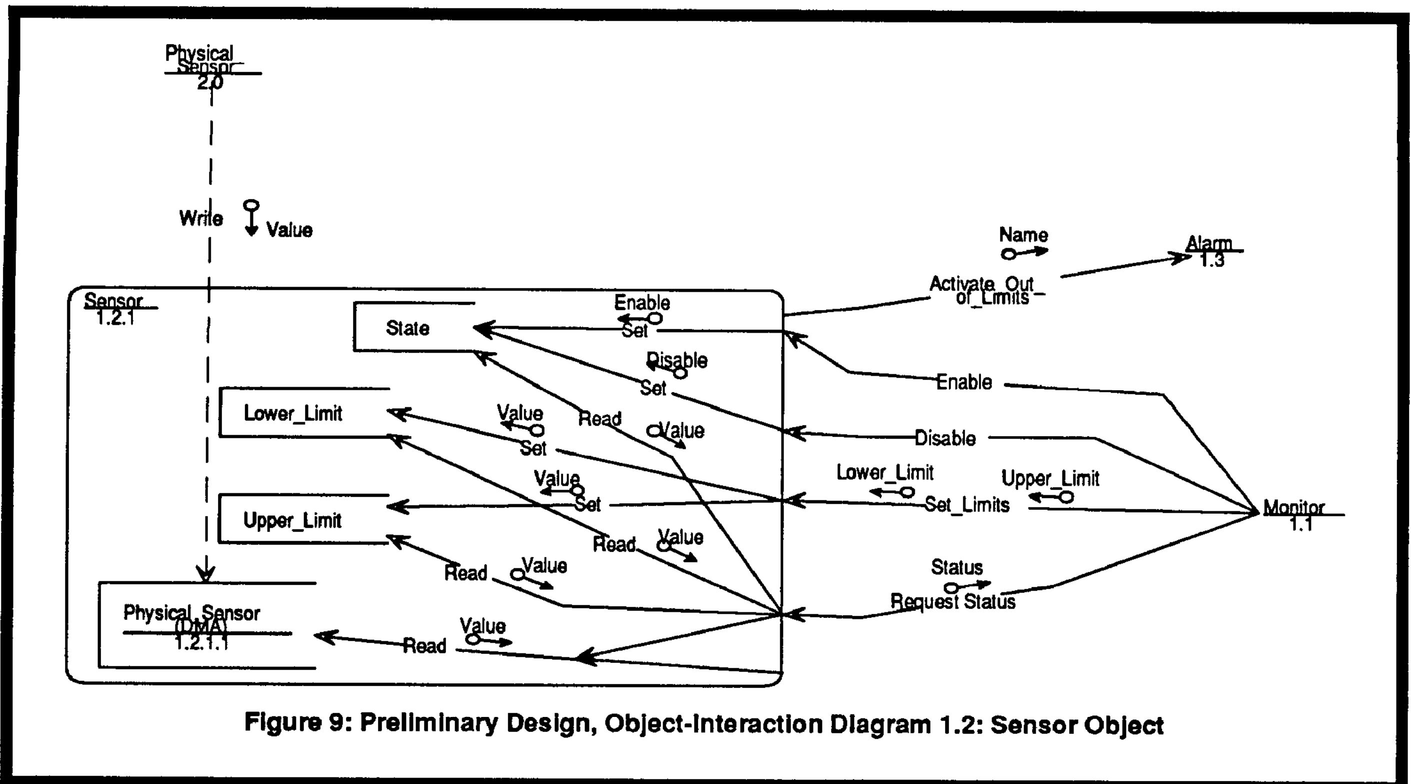


Figure 9: Preliminary Design, Object-Interaction Diagram 1.2: Sensor Object

is a straightforward mapping from the objects in Object–Interaction Diagrams to the Ada software components. Generally, in creating the Ada representation, a graphic representation is produced first, then the textual representation.

Here is the process of representing the objects with Ada software components.

1) Represent each object in the Object–Interaction Diagram (see Figures 2a–b and 11a–b).

   a) Represent each active object as an Ada subsystem, package, or task.

   b) Represent each passive object as an Ada object or package.

   c) Localize each Ada representation to the appropriate program unit or subsystem.

2) Represent each class in the Object–Class Diagram (see Figures 5 and 11a–b).

   a) Represent each class as an Ada type, an "Abstract Type"[4], or a generic package.

   b) Localize each representation as appropriate to a program unit or subsystem.

[4] An "Abstract Data Type" package or an "Abstract Task Type" package.

3) Represent each interaction between objects in the Object–Interaction Diagram (see Figures 2a–b and 11a–b).

   a) Represent each operation in the Object–Interaction Diagram as a subprogram or task entry in the appropriate package or task.

   b) Represent the initiation of the operation as a call to the appropriate subprogram or entry.

   c) Represent each object and data flow as a formal parameter in the appropriate subprogram or entry.

   d) Represent each object and data flow as an actual parameter in the call to the appropriate subprogram or entry.

   e) Represent each error flow as an exception on the appropriate call and define the exception in the appropriate program unit.

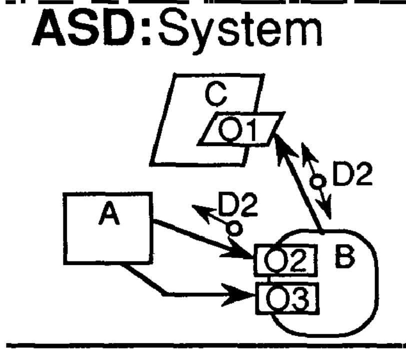
4) Review the State–Transition Diagram for details of behavior that affect the Ada structure (see Figure 11b).

   a) If the initiation of operation by an object is in response to an operation on the object, then associate the call to the object

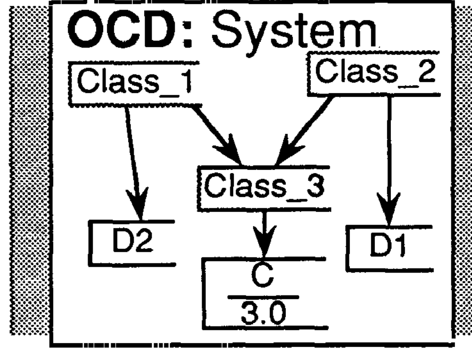
   b) If an active object "waits" for the initiation of one of a group of operations on it, then add a selection box around the corresponding entries.

5) Add Ada–specific components needed to complete the representation.

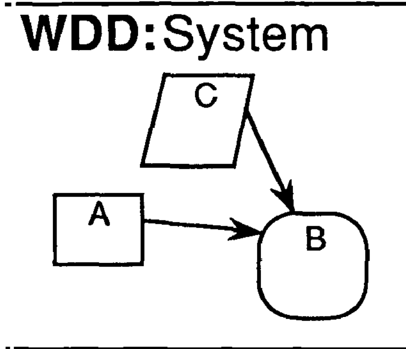
S/W Architecture Specification



Object/Class Specification



Behavior Specification



PDL: System

```
procedure System
is
  ...

begin
  ...
end System;
```

Attribute Specification

AT: System

	Scale	Test	Worst	Planned	Record	Past
Workability						
Reusability						
Portability						
Performance						
Useability						

Summation of Results

AD

	Subsystem	Description	Entity	Category	Object Class	Notes
A						
B						
C						
D1						
D2						

Figure 10: Detailed Design Activities & Representations

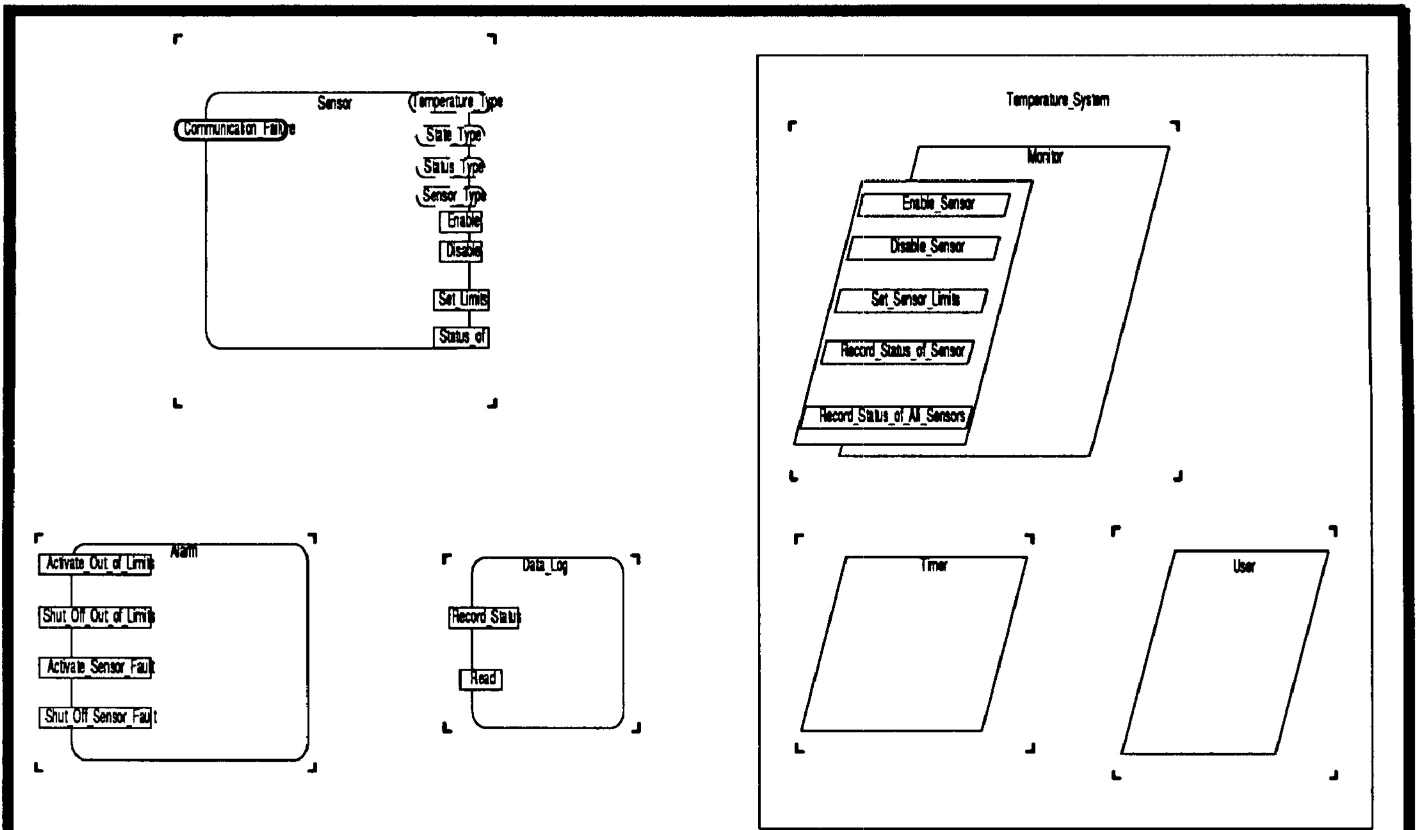


Figure 11a: Ada Structure Diagram 0: Temperature_Monitor_System Object

Sensor
Temperature_Type
Communication_Failure
State_Type
Status_Type
Sensor_Type
Enable
Disable
Set_Limits
Status_of
Monitor
Collection_of_Sensors
Enable_Sensor
Disable_Sensor
Set_Sensor_Limits
Record_Status_of_Sensor
Record_Status_of_All_Sensors
Collection_of_Sensors (Sensor_Name)
Communication_Failure
Upper_Limit
Lower_Limit
Collection_of_Sensor (Sensor_Name)
Sensor_Name
Alarm
Activate_Out_of_Limits
Shut_Off_Out_of_Limits
Activate_Sensor_Fault
Shut_Off_Sensor_Fault
Data_Log
Record_Status
Read

Figure 11b: Ada Structure Diagram 1.2: Monitor Object

The process of representing the objects as software components can be partially automated. However, the designer needs to apply his knowledge of the language in order to choose effectively between alternate language representations of the some objects (see Figures 11a,b). The generation of the Ada PDL from the graphics and vice versa has been automated.

Object–Class Specification

The Object–Class Specification identifies the classes (types and generics) of the software objects in the architecture, and the relationships between the classes, using the Object–Class Diagram. This is principally documentation maintenance.

Behavior Specification

The Behavior Specification activity represents the dynamic behavior of the language–specific architecture, and demonstrates that the behavior of the architecture implements the behavior specified for the model. A program design language for the implementation language (e.g., Ada PDL) is used to represent the behavior of the software components textually (other graphic and textual representations could be used).

The representation of the dynamic behavior of the software requires the representation of the behavior of each software component. Continuing to use Ada as an example, in Ada the components are packages, tasks, objects, or abstract types. Behavior Specification will:

1) Represent the external view of the behavior of the corresponding object or class of objects.

   a) In the Ada PDL, add a comment to the declarations of the Ada software component describing the behavior of the component from the perspective of the user of the component.

   b) Add comments to the declarations of the subprograms or entries in the Ada software component describing the subprograms or entries from the perspective of their user.

2) Represent the internal view of the behavior of the corresponding object.

   a) Create a body for each Ada software component, and describe in the Ada PDL the implementation of the component in terms of the internal behavior of the objects.

   b) Create a body for each subprogram or entry in the component, and describe its implementation in terms of the effect of the operation it represents.

3) Add descriptions of Ada–specific components needed to complete the representation.

To demonstrate that the correct dynamic behavior has been described for the software architecture, OOSD demonstrates that the dynamic behavior described for each software component is correct. This demonstration requires the analysis or execution of the software component to verify that it behaves according to the description of the behavior of the corresponding object. This process could be automated (for example, executing the Ada PDL).

Attribute Specification

The Attribute Specification activity identifies the attributes of each software component, and demonstrates that the attributes of the software components meet the required attributes of the corresponding objects. The Attribute Specification Forms and Tables are used to associate the corresponding attributes of objects and components. The components must be analyzed or executed to verify that they achieve the required attributes.

Object–Oriented Design Results

OOSD design results in a complete software architecture corresponding to the model of the problem, describing "how" to implement a solution to the problem by means of a set of software components whose interactions, behavior, and attributes represent the corresponding objects in the model. The software architecture is shown by the graphic and textual views described in the preceding sections. In addition, a Software Architecture Dictionary is refined from the System Requirements Dictionary for summarization and cross–reference.

IS IT ANALYSIS OR DESIGN?

The historical distinction between requirements analysis and design is that requirements analysis describes *what* needs to be done, while design describes *how* to do it. This distinction has been described as "valid but infuriating" [1]. Lately some have argued that there is *no* distinction between requirements analysis and design. Note, however, that

1) many problem requirements are expressed by describing a solution (e.g., "we want a user–friendly interface" is expressed as "we want a mouse–driven window environment") [6]; and

2) the maxim "One man's ceiling is another man's floor" applies to requirements and design (e.g., a missile is "design" to the person who wants a delivery system, and "requirement" to the person who creates the missile).

Conceptually, requirements analysis can be said to end when the required context, behavior, and attributes of a problem have been clearly identified ("what"). The next step beyond enters preliminary design ("how"). However, often managerial or technical factors (e.g., procurement requirements, separation of system analysis from software analysis) call for top–level decomposition of the system during requirements analysis. Since these factors occur frequently in current projects, the requirements–analysis portion of OOSD has been described as including top–level decomposition. However, OOSD can be applied equally well with other definitions.

PROBLEMS OF SIMILAR METHODS

Two particular methods have achieved some repute for

object-orientation, each of which essentially proceeds by adding object orientation onto Structured Analysis. Shlaer and Mellor [10] have developed an "Object-Oriented Analysis" method (OOA) which has as its goal the development of an "information model" of the problem. They also address the transition from the information model to a software architecture. The highest-level classes of objects[5] in a problem description are represented in this information model. Each class is represented by its attributes and its processing states. The states are decomposed by function, using data flow diagrams. Objects are entries in a table of attributes which is defined for each class. The system specifications developed using OOA reflect a functional modularity rather than a structural and behavioral modularity of objects. In this respect, OOA is not very different from Structured Analysis, since most "data flow diagrams contain all the meaningless pseudo-action words that structured analysis purists warn against" [1], especially at the top levels of the data flow diagrams. As a result, combining OOA with object-oriented design yields many of the same problems observed in combining Structured Analysis with object-oriented design. The authors' work on identifying classes and objects does facilitate the transition process, which in their method is still necessary.

Bailin among others [1] has developed an "Object-Oriented Specification" method (OOS). In this method, the top-level entities[6] and functions are represented in a "Entity [Object] Data Flow Diagram" (EDFD). These entities and functions are identified by creating a Structured Analysis DFD (the author's description leaves the level of detail of the SA DFD unclear). The noun in the process name is extracted to identify a top-level entity, and the verb in the process name is extracted to identify a function (e.g., create_*message* identifies the entity "message" and the function "create"). Functions in the EDFD "must occur in the context of an entity"; each entity is decomposed into its component entity and functions, *and* each function is decomposed into subfunctions. This function context rule and decomposition method together achieve a degree of structural and behavioral modularity. However, decomposing functions into their subfunctions causes Bailin's definition of an entity to include functions which the entity performs on other entities. Also, Bailin reports that as a result of a decision to "subordinate control flow to data flow", he does "not see entities as objects with well-defined interfaces." Essentially OOS does not eliminate the transition between a Structured Analysis view of the world and an object-oriented view. However, the transition is performed earlier in the software development process, when it may be easier to accomplish.

The difficulties of these methods result from representing problems by multiple abstract models based on different concepts. During Structured Analysis, two to four models are applied (depending on method): an information model, a data flow model, a process model, and a control model. Object-oriented design activity then introduces additional models. In OOSD, the object model has a substantially different focus, and integrates the information which Structured Analysis assigns to separate models.

TOOL SUPPORT AND NOTATION

A number of companies have integrated OOSD in their software development environment using commercial graphic-design tools. The projects using OOSD have adapted the notation on occasion to conform with pre-existing tool conventions. Projects using Adagen® are able to use the notation shown in the present paper. Those using Cadre Team*work*® use a notation which conforms to Cadre's current editors. Both tools provide capabilities which facilitate OOSD application, such as Adagen's capabilities for linking graphic and textual representations, generating code frames, and generating diagrams from code ("reverse engineering"); and Cadre's capabilities for managing representations, creating name dictionaries, and generating code frames. OOSD is being adapted to other tools.

The notation used in OOSD has evolved rapidly over the last year and a half. The user community is very active in continued participation in OOSD development, which has been particularly helpful in adapting representational views to equivalent notations supported by other tools and to object-oriented languages, as well as in the evolution of the method and its underlying concepts.

CONCLUSION

The Object-Oriented Software Development Method (OOSD) differs from Structured Analysis, and as examination shows from other current methods which are partly object-oriented, by developing a single, consistent abstract model of the elements of a problem. This model is used during requirements analysis to identify the "what", and during design to determine the "how". During requirements analysis OOSD develops its model, which identifies the required objects, classes, functions, behavior, and properties (attributes) of the problem. During design, the model is refined into an architecture for software components, with a smooth transition to code.

Identifying the objects in a problem is the hardest part of this or any object-oriented method. OOSD contributes to relieving the burden with the definition of "object", and the uniform treatment of active and passive objects, set forth above. Currently, the identification process cannot be automated: it takes the application of one or more skilled human beings who understand the problem domain and the concept of objects. This, of course, is parallel to the requirements of Structured Analysis upon talented individuals. The crux of development has been shifted from identifying functions and data flows to identifying objects. Users have found this different approach applies well to real-world problems.

[5] They use software engineering terms in unconventional ways; e.g. what they call object most others call class, and what they call instance of an object most others call object.

[6] The term entity is synonymous with object and is used during requirements analysis.

® Adagen is a registered trademark of Mark V Systems Limited. Teamwork is a registered trademark of Cadre Technologies, Inc.

REFERENCES

1. Bailin, S.C., "An Object-Oriented Requirements Specification Method"; *Communications of the ACM*, New York, NY, Volume 32, Number 5, pp. 608–32 (May 1989).

2. Booch, G., *Software Components with Ada*, The Benjamin/ Cummings Publishing Company, Inc., Menlo Park, CA (1987).

3. Booch, G., *Software Engineering with Ada*, The Benjamin/ Cummings Publishing Company, Inc., Menlo Park, CA, pp. 287–88, 299, 301 (1st ed. 1983)(the substantially improved treatment of this problem at pp. 334–54, 2nd ed. 1987, is less interesting for the present purpose).

4. Buhr, R., *Machine Charts for Visual Prototyping in System Design*, SCE Report 88–2, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ont., Canada (August 1988).

5. Buhr, R., *System Design with Ada*, Prentice–Hall, Inc., Englewood Cliffs, NJ (1984).

6. Gilb, T. and Finzi, S., *Principles of Software Engineering Management*, Addison–Wesley Publishing Co., Reading, MA (1988).

7. Nielson, K. and Shumate, K., *Designing Large Real–Time Systems with Ada*, Multiscience Press, Inc., McGraw–Hill Book Co., New York, NY (1988).

8. Orr, K., *Structured Requirements Definition*, Ken Orr and Associates, Inc., Topeka, KS (1981).

9. *The Random House College Dictionary*, Random House, Inc., New York, NY, pp. 339, 916 (rev. ed. 1975).

10. Shlaer, S. and Mellor, S.J., *Object–Oriented Systems Analysis: Modeling the World in Data*, Prentice–Hall, Englewood Cliffs, NJ (1988).