Requirements Analysis with the Object–Oriented Software Development Method

by Edward Colbert
Absolute Software Co., Inc.

By integrating object–oriented requirements analysis with OO design, the Object–Oriented Software Development Method (OOSD) allows a practical focus on the objects of a problem throughout. OOSD requirements analysis clearly represents problem requirements and leads smoothly to design, creating a single consistent abstract model with strong validation and verification, which is easy to maintain, and closely follows the "real world".

INTRODUCTION

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, behavior, and properties, in a single consistent abstract model. In requirements analysis, this model identifies the "what": the required objects, classes, functions, behavior, and properties 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.

OOSD formally defines the properties of objects, describing the system as an object. The system is then refined into its component objects. Classes are methodically identified by generating them from objects in the system. A particularly thorough verification procedure establishes that the system is correctly implemented and achieves the required properties. Objects are treated uniformly, including the system object. By performing the same essential activities in analysis, preliminary design, and detailed design, an unusually consistent model is built, which closely follows the "real world", can be tested early, and is easy to modify and re–use.

OOSD can be applied in a variety of development models including evolutionary, spiral, waterfall, prototyping, and market–driven (Royce 1970; Boehm 1981; Boehm 1988; Gilb 1988; Sodhi 1991; Summerville 1992). In this paper I will use the waterfall model, which is relatively linear and suitable for written exposition. I will discuss OOSD requirements analysis, with an overview of how OOSD is applied to design.

OOSD REQUIREMENTS ANALYSIS

Our goal during requirements analysis is to understand "what" the customer "needs" us to build. We build a model of the application and validate that our model will actually meet the customer's needs. OOSD recognizes the system itself as an object.

In OOSD requirements analysis, four activities are applied to the system object to create the model of the application. Graphic and textual representations produced by these activities are used to validate the model. The four activities are

1) Object–Interaction Specification
2) Object–Class Specification
3) Behavior Specification
4) Property Specification

These activities can be performed in almost any order once an initial Object–Interaction Specification has been done (and will be discussed in a different order below). An iterative approach is recommended, since the performance of each activity generally suggests refinements in the others.

Published in substantially the same form in A. Carmichael ed., *Object Development Methods* (SIGS Books 1994).

Two different strategies for requirements analysis have been found to be useful. When starting a new development ("green–field development"), we begin by identifying the objects and classes that are required in the context of the system object. When reworking an existing system (re–engineering a system, or restructuring an organization or business), we apply an approach described by Ken Orr (Orr 1981) as "Middle–Up–Down", where we describe the objects and classes that are components of the current system or organization ("middle"), then abstract away the details of the implementation and define the boundary of the new system or organization ("up"), then design the new components ("down").

In this paper I will show how OOSD is used in a new development, analyzing a temperature–monitoring problem given in an early book by Booch (Booch 1986).[1]

Temperature Monitoring Problem

There exists a collection of ten independent sensors that continually monitor temperatures in an office building. Initially, all of the sensors are disabled. We may explicitly enable or disable a particular sensor, and we may also force its status to be recorded. Furthermore, we may set the lower and upper limits of a given sensor. In the event that any of the enabled sensors register an out–of–limits value, the system must immediately post an alarm condition. Additionally, it must request and record the status of all the sensors every 15 minutes (set by a timer hardware interrupt). If we do not get a response from any sensor within 5 seconds after this time we must assume that the sensor is broken and immediately post another alarm. Asynchronously, we may get a user command to enable or disable a specific sensor, set the temperature limits, or force the status of a given sensor to be recorded. In any case, failure of the user interface must not affect the monitoring of any currently enabled sensors.

The sensors are located in the lobby, main office, warehouse, stock room, terminal room, library, computer room, lounge, loading dock, and cleaning room. They use memory–mapped I/O ports to write integer values to the ten 16–bit words starting at address 16#0100#; each value when multiplied by the accuracy of the sensor (0.5°C) results in the external temperature value. When a fault is detected or a sensor goes out of limits, the system will turn on a warning light, which the user must manually clear. To activate a warning light, the system must place all 1's (16 bits) in the memory–mapped I/O ports: 16#0010# — address of fault warning; 16#0011# to 16#001A# — address of out–of–limits warning.

Object–Interaction Specification

Object–Interaction Specification identifies the objects which are required to communicate with our system, and the interactions between our system and these objects, using an Object–Interaction Diagram and a Dictionary. Object–Interaction Diagrams (OID) describe a set of objects and the interactions between the objects (see Figure 1). The OID is the fundamental representation of the model of the application; the top–level OID, called a context diagram or external view, describes the system object in the context of those objects, outside the system, which the system object interacts with. The Dictionary lists every part of the system model, e.g. objects, classes, interactions, giving the purpose, requirements, and any other useful information.

Object–Interaction Diagram

In an OID (see Figure 1), active objects are shown by round–cornered rectangles (e.g. *Temperature_ Monitor_System*), passive objects are shown by square–cornered rectangles (see Figure 13), and external

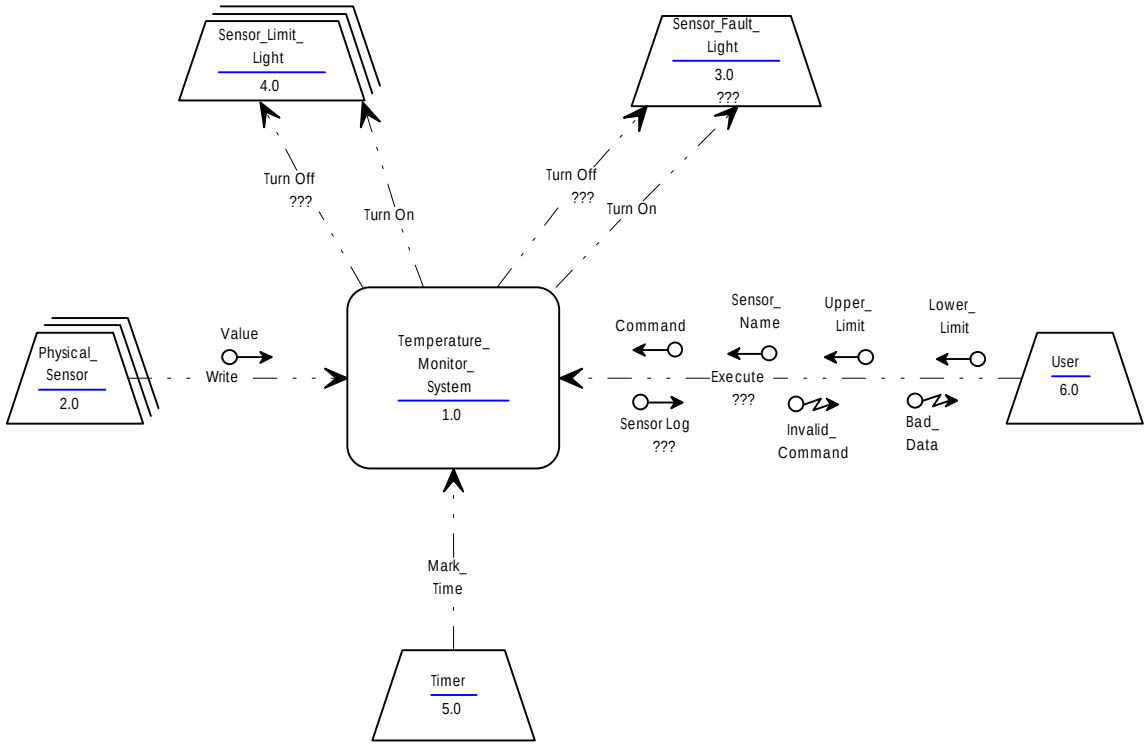
[1] Although a later edition of the book has appeared, the problem as stated in the edition I cite has certain useful features for this paper. The discussion below will find various inadequacies in the problem statement. This process is typical of software development; it is nearly inevitable, given human limitations, and should not be seen as a reflection on Booch.

objects are shown by trapezoids (e.g. *User*). "Shadowing" (e.g. *Physical_Sensor*) indicates multiple objects of the same class which are distinguished by some form of mapping, e.g. an address. The names of objects should be nouns.

> OOSD considers an object *active* if it displays independent motive power; if not, *passive*. A *passive* object can only perform work, and can only change its state, while performing one of its operations in response to a request by another object, ultimately under the motivation of an active object. An *active* object need not exercise its motive power (for example, a human being sometimes acts in response to requests or commands). A printer demon would be represented as an *active* object, even though it does not perform any work until it receives a print request from another object; when the demon receives the request, the demon releases the other object and performs the actual printing on its own power. *External* objects are objects outside the scope of the system to be implemented.

An interaction describes a request by one object to another that the requested object perform one of its operations, along with any objects or data that are exchanged as part of the interaction. Interactions are shown in an OID by arrow–shaped arcs connecting the objects that interact. Interface interactions, i.e. with an external object, are shown by lines with a dash–dot–dot pattern. Often their implementation involves hardware and has special properties. The arc is labeled with the name of the operation being requested, which should be a verb or verb phrase.[2] An operation is **requested by** the object shown at the tail of the arrow in the interaction, **performed by** the object shown at the head of the arrow.

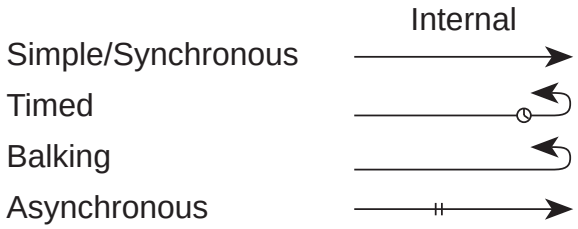
Figure 1: Context Object–Interaction Diagram for the *Temperature_Monitor_System*



2 If this convention is not strictly enforced, people accustomed to Structured Analysis tend to treat requests as data flows, and people accustomed to languages using a message and method scheme, like SmallTalk (Goldberg, 1989; Wegner, 1990), tend to treat requests as messages. OOSD views "methods" as ways of implementing operations, and "messages" as ways of implementing requests.

The arc is annotated to distinguish four kinds of requests, according to the way the request is issued (Figure 2). In a **synchronous** request, the two objects will remain synchronized throughout the interaction (like a telephone conversation).[3] In an **asynchronous** request, the requesting object issues its request, but does not wait to see if the request is received or the requested operation performed (like mailing a letter). **Timed** and **balking** requests are essentially synchronous, but the requesting object can "give up", i.e. stop waiting. In a timed request, the requesting object gives up if the performing object does not start performing the requested operation in a specified period of time. In a balking request, the requesting object gives up if the performing object does not start performing the requested operation immediately. In this paper I take up only synchronous requests. The other types are typically introduced during the refinement of analysis and design.[4]

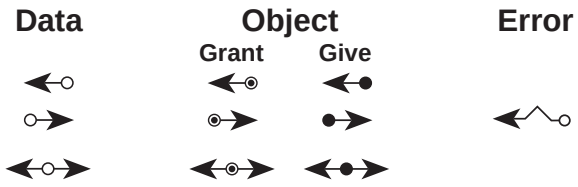
Figure 2: Types of Request



Four kinds of arrows show exchanges of objects or information during an interaction. The arrowhead on the couple indicates the direction of flow. (See Figure 3.) OOSD distinguishes **data** flows, **grant object** flows, **give object** flows, and **error** flows. For example, in Figure 1 the *Execute* interaction between the *User* object and the *Temperature_Monitor_System* object consists of the *Execute* operation request; 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 requested by the *User* object, and performed by the *Temperature_Monitor_System* object.

By **data flow**, OOSD means an transfer of information between two communicating objects about one or more objects. By **object flow**, OOSD means that one of the communicating objects actually hands over a measure of control of some object, either granting access only while retaining ultimate control (**grant object**) or giving over control entirely (**give object**). An **error flow** is a kind of data flow reporting that an object failed to perform a requested operation, with the reason. The error flow is given a name that describes the failure. Other data flows, and object flows, are given the names of the data or objects to be transferred, from the perspective of the performing object.

Figure 3: Flows



In all OOSD diagrams, question marks signal an issue that needs to be discussed with the customer (or system engineer). The issue is described in the Dictionary entry for the entity marked with a question.

---

3 If either object is internally concurrent, then possibly only a component is actually synchronized with the other object; if both are internally concurrent, only components of each may be synchronized. This will be clarified by the behavior descriptions of each object.

4 OOSD users sometimes call a synchronous request "simple", because its semantics are equivalent to a message in SmallTalk, a member function call in C++, or a subprogram or entry call in Ada. An asynchronous request needs some form of mailbox.

Identifying Objects and Interactions

Many object–oriented methods which identify objects during requirements analysis perform what Shlaer & Mellor call an "object identification blitz" (Booch 1991; Coad and Yourdon 1991; Rumbaugh, Blaha et al. 1991; Shlaer and Mellor 1992). In OOSD requirements analysis, we consider the system as an object, and we only need to identify the other objects which are either required to communicate with the system, or are exchanged (or about which data is exchanged) during interactions. For example, in Figure 1 the *Temperature_Monitor_System* is our system object, *User*, *Hardware_Timer*, *Physical_Sensor(s)*, *Sensor_Limit_Light(s)*, and *Sensor_Fault_Light* are all objects which communicate with the system; and *Command*, *Sensor_Name*, *Lower_Limit*, *Upper_Limit*, *Sensor_Log*, and *Value* are all objects about which data is exchanged during interactions.

We should be able to identify the interactions between the system object and external objects from what is said, in whatever oral or written statements we are given, about communications between objects. For example, we were given the requirement "when a fault is detected or a sensor goes out of limits, the system will turn on a warning light." We represented this as the two interactions where the *Temperature_Monitor_System* requests the *Turn_On* operation of a *Sensor_Limit_Light* or of the *Sensor_Fault_Light*. Interactions between external objects are by definition generally outside our problem.

In deciding what external objects and interactions to represent, we need to balance two concerns. We want a complete and consistent model, and if our model lacks these qualities it will probably not be correct (see "Product of Requirements" section below. We also want to represent exactly what the customer specified, so we can discuss our model with the customer. In OOSD, we strive for these potentially contradictory goals by adding objects or interactions to those in the problem statement as needed for completeness, flagging new or inconsistent objects and interactions with question marks, and annotating our questions in Dictionary entries.

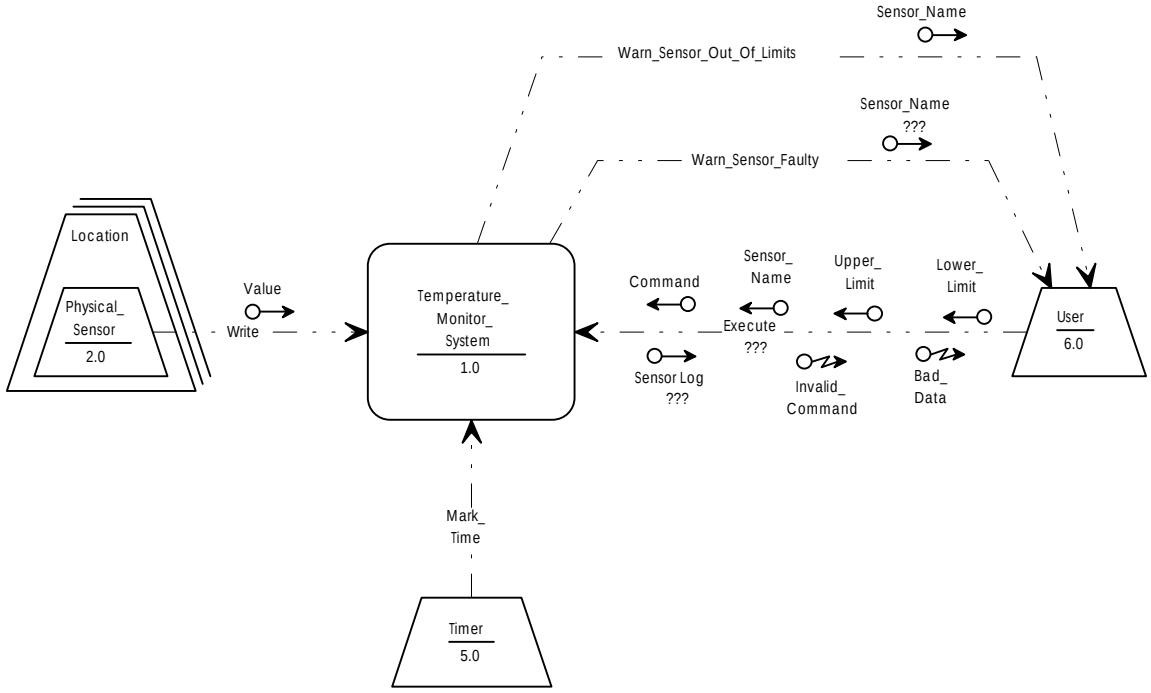
For example, our problem statement said "the system will turn on a warning light which the user must manually clear." Plainly the *Temperature_Monitor_System* should request the *Turn_On* operations of the *Sensor_Limit_Light(s)* and the *Sensor_Fault_Light*, but should it request the *Turn_Off* operation? Turning off (clearing) the lights was not one of the user commands given in the problem statement. Also, we are told of multiple *Sensor_Limit_Light(s)*, but only 1 *Sensor_Fault_Light*. In our model (Figure 1) we included 2 interactions showing the *Temperature_Monitor_System* requesting the *Turn_Off* operation of both *Sensor_Limit_Light(s)* and the *Sensor_Fault_Light*. We noted the 2 interactions and the *Sensor_Fault_Light* with question marks, and described our question in the dictionary entries for the *Turn_Off* interactions and the *Sensor_Fault_Light*.

In representing external objects and interactions, any given diagram should be kept at a consistent level of abstraction, from the perspective of our system. Usually the problem statement will be found to contain one or more design decisions expressed as requirements. For example, our statement says "the system must post an alarm condition" when a sensor registers an out–of–limits value, but this is really a design decision implementing a requirement that the system warn the user when a sensor is out of limits. Furthermore, we are given a requirement that "posting an alarm condition" be implemented by "turning on a warning light", and that "turning on a warning light" be implemented by "placing all 1's (16 bits) in a memory–mapped I/O port". While we must represent what the customer specified, we must also determine the level of abstraction we currently need to work at, and stick with it so long as we are there. If some other level of abstraction appears useful, it should be given another diagram, and possibly deferred to a later activity.

In Figure 1, we have represented *Sensor_Limit_Light* and the *Sensor_Fault_Light* as external objects, and have shown the *Turn_On* and *Turn_Off* interactions. To be consistent, we should show a user–interface device rather than the *User*, and we should show how the device interacts with the system. However, since no particular device was required, this ideally should be deferred to design. Figure 4 is a view of the system at a more uniform level of abstraction. We have abstracted away the lights, and represented the

Warn_Sensor_Out_Of_Limits and *Warn_Sensor_Faulty* interactions between the *Temperature_Monitor_System* and the *User*. At the other extreme, we could draw another diagram showing the memory–mapped I/O ports instead of the lights as external objects, and representing *Write* interactions between the *Temperature_Monitor_System* and the ports. In this diagram, we would also represent the sensor ports, which were specified in the problem statement, and ports needed to support the user–interface device, which were not specified. Such a diagram would obscure "what" we expect to accomplish, e.g. warning the user that a sensor is faulty or out of limits, while clarifying "how" we expect to accomplish it, e.g. by writing to certain ports.

Figure 4: Context Object–Interaction Diagram for the *Temperature_Monitor_System* — Problem View



As a general rule, I recommend drawing both a "Problem View" of the context of the system, e.g. Figure 4, as a requirements–analysis activity, and drawing a "System Engineering View", e.g. a refined version of Figure 1, as a design activity. The Problem View better illustrates the communications that must occur, while the System Engineering View better illustrates the devices, such as the lights in our example, used to implement the required communications. In addition, because the Problem View is more abstract, it leaves us more flexible to meet system–engineering design changes, and is more re–useable and portable.

In most cases the requester and the performer of an interaction, and thus its direction, are obvious from the problem statement. Sometimes the direction is not obvious, especially during design, or during requirements analysis if the context of the system has not been well defined. We then select the direction that seems most natural, and study the impact of the choice. Usually setting one direction instead of the other will simplify the behavior of the two objects, or otherwise promote software engineering goals. For example, one direction may make polling unnecessary, improving the efficiency of one or both of the objects, or ease adding operations to one of the objects, making it more adaptable.

Objects which only request services from the system are not as critical to identify as the services they request. The system does not care what object makes the request; as long as the proper information or objects are supplied, the system will perform the service. However, showing the objects that issue requests

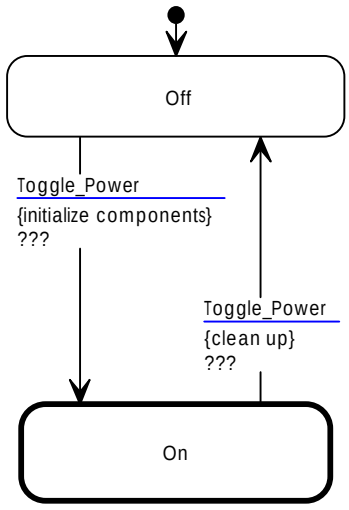
is useful for talking to the customer or system engineers, and facilitates communication among reviewers and developers.

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 1 does not show what conditions cause *Temperature_Monitor_System* to request *Turn_On* from *Sensor_Fault_Light*.

The Behavior Specification activity identifies the **dynamic** behavior of the system object. OOSD uses statecharts (Figures 5 and 6), derived from Harel (Harel 1987; Harel 1988), to represent behavior; however, other graphic representations could be used.

Figure 5: Top–level Behavior Diagram for the *Temperature_Monitor_System* (External View)



Harel Statecharts are based on traditional state transition diagrams which describe "behavior in ways that are clear and realistic, and at the same time formal and rigorous, sufficiently so to be amenable to detailed computerized simulation [or other analysis techniques]" (Harel 1987). The behavior of a system is described in terms of states, stimuli which cause changes of state, and any actions that are performed in response to a stimulus in addition to the change of state. Harel extended traditional state transition diagrams by formalizing the representation of hierarchical states ("depth"), i.e. states nested within other states, simultaneous ("orthogonal") behavior, and communication between simultaneous behaviors. As Harel puts it (Harel 1987):

> In a nutshell, one can say:
> statecharts = state–diagrams + depth + orthogonality + broadcast–communication

OOSD Statecharts integrate Harel's basic notation with the object–oriented concepts of objects, classes, and operations, and can be used to describe any object.

OOSD Statechart Notation

In an OOSD Statechart, a *basic state* is represented by round–cornered rectangles (e.g., *Off* and *On* in Figure 5). The label of a basic state contains the name of the state, and a description of any work which is performed while in the state (except work that for accuracy must be shown by decomposing the state). If work is mentioned, its description is placed under a horizontal line.

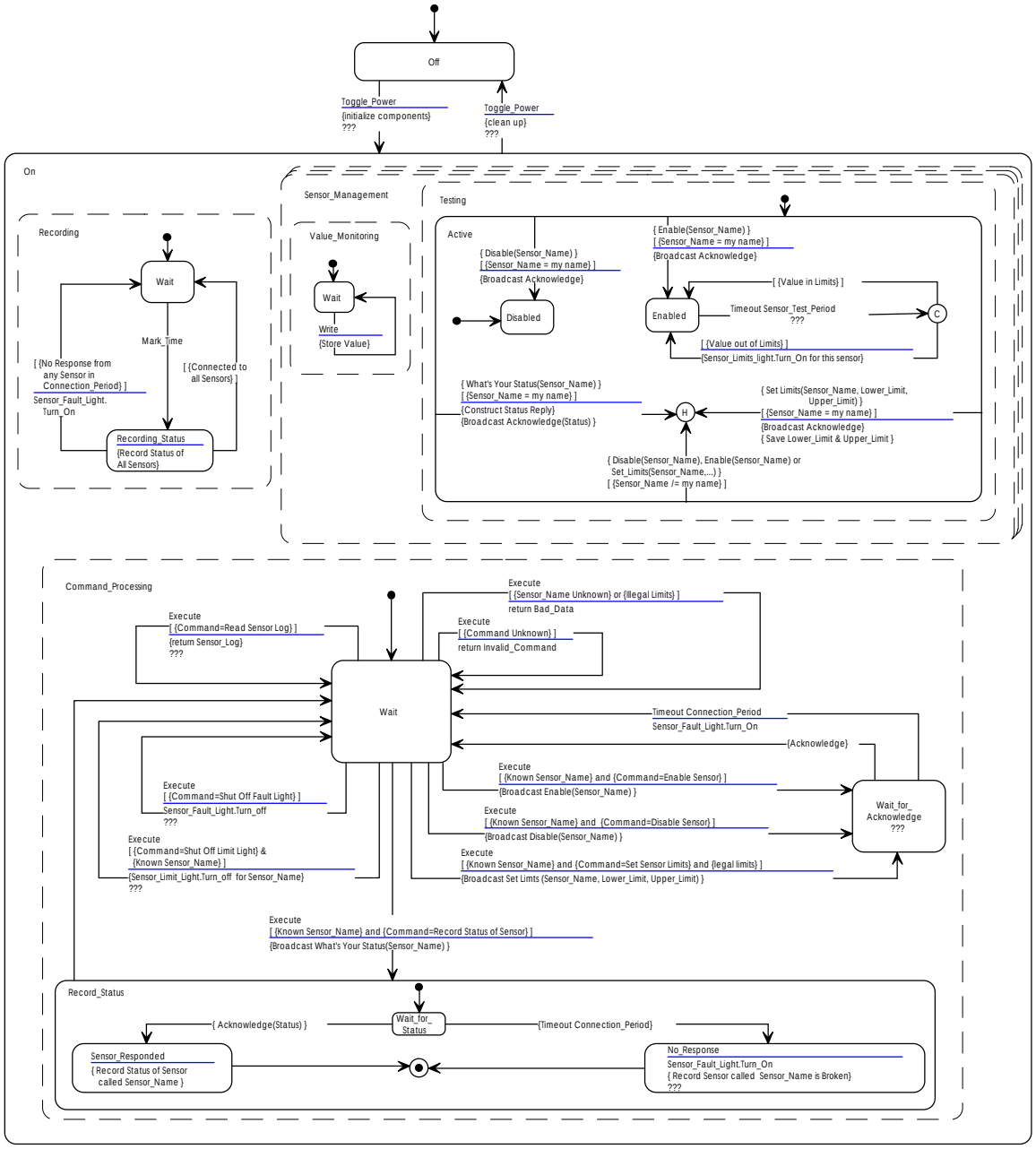
A state may summarize more detailed behavior, in which case it is drawn with boldface lines when collapsed, and normal lines when expanded (e.g., the *On* state in Figures 5 and 6). A state may have substates, shown by drawing their rectangles inside its rectangle. A state may also be drawn with separate, dashed rectangles to represent *orthogonal components*.[5] A state (and any orthogonal component) may contain substates, or orthogonal components, but not both.

---

5 Harel draws dashed lines across the rectangle to show these components, but drawing the components as separate parts, so they can be treated independently (such as by "selecting" them with drawing programs), makes it easier to manipulate the diagram.

In Figure 6, we have expanded the *On* state of the *Temperature_Monitor_System* and all of its substates and orthogonal components, illustrating how the system behaves when it is on. The *On* state has three orthogonal components, indicating that when the *Temperature_Monitor_System* is *On*, it simultaneously exhibits *Recording*, *Sensor_Management*, and *Command_Processing*, and that *Sensor_Management* simultaneously exhibits *Value_Monitoring* and *Testing*. The "shadowing" on the orthogonal component *Sensor_Management* indicates multiple occurrences of this behavior; the number of occurrences is part of the dictionary entry for *Sensor_Management*. *Command_Processing* consists of 3 states, *Wait*, *Wait_for_Acknowledge*, and *Record_Status*. *Record_Status* has 4 substates, *Wait_for_Sensor*, *Sensor_Responded*, *No_Response*, and a *termination state* represented by a bull's–eye.

Figure 6: Expanded Behavior Diagram for the Temperature_Monitor_System (External View)



In representing behavior, we first note the state in which the behavior starts. The diagram shows a solid circle called the *start* or *default marker*. A change–of–state arc points from the start marker to the *start state*. In Figure 5, *Off* is the start state.

A change of state, called a *transition*, is represented by an arc connecting the states. Most changes of state are conditional; the condition is described in a label on the arc. If some work is to be performed along with the change of state, the work is described in the label, below a horizontal line separating it from the condition (even if there is no condition, the horizontal line is always shown). Any action shown in the label is performed, followed by any work of the state. If a change is to occur by default, i.e. after completing any work of the state if no condition for another change is met, no condition is described. In our notation, an object remains in a state until some condition for change is met (including a default transition). If for a particular state and condition, the result of the condition's occurring will be a return to the same state, OOSD represents this explicitly by drawing a "transition" that returns to the state it came from.

In many cases, behavior ends in a state from which there can be no change. Such a state is called a *termination state* and is represented by a bull's–eye. We chose to show a possible cycle from *Off* to *On*, instead of showing a termination state, pending clarification of the requirements for turning the system on and off.

Figure 5 shows that the system will change from *Off* to *On* when a *Toggle Power* request is received from some other object, and when the request arrives, the system will *initialize (its) components*. We have placed question marks on the 2 transitions because the problem description did not specify how the system is to be turned on or off, nor any requirements related to the often critical issues of initialization and clean–up.

We consider that a *condition* for a change of state will be some *event*, or the result of some *internal test*, or both. We consider that an *event* is (a) the receipt of either a request from another object that the object perform one of its operations, or a message from another orthogonal part, or (b) a time–out. An *internal test* may (a) test what state another orthogonal part is in, (b) test the value (state) of a parameter of an operation, or (c) test an attribute of the object whose behavior we are representing. An internal test is shown in a label by brackets.

In Figure 6, each of the three transitions from *Command_Processing.Wait* to *Command_ Processing.Wait_for_Acknowledge* shows a change conditional on both an event and a test result, e.g., the system will change when *Execute* is requested and the *Command* is *Enable Sensor* and the *Sensor_Name* is known.

When a change leads to a state that has substates or orthogonal parts, we must show clearly which substate is the *current* state, which we specify in three ways: (a) the transition can be directly attached to a specific substate, (b) a *default* (*start) state* can be specified, (c) a *state entrance* indicator can be specified.

In Figure 5, transitions are drawn from *start markers* in each orthogonal component, which shows that when changing to the *On* state, the *Recording*, *Command_Processing*, and *Sensor_Management.Value_ Monitoring* orthogonal components each start in their respective *Wait* substates, while *Sensor_Management.Testing* starts in its *Active* substate.

When a change leads out of a state that has substates or orthogonal parts, there is a "change out of all substates". In Figure 6, the transition from *Sensor_Management.Testing.Active* to *Sensor_Management. Testing.Active.Disable* indicates that the system will change to the *Disable* state from the current substate of the *Active* state when a *Disable(Sensor_Name)* message with the appropriate *Sensor_Name* is received.

We consider that work consists of one or more *actions*,[6] and that if more than one action is specified, they are to be performed in the sequence stated. We consider that an *action* is (a) a request that another object perform one of its operations, (b) a message broadcast from the orthogonal component we are looking at to others, or (c) an "informal description", i.e. a relative abstraction that may later (possibly not until design) need further refinement. Bearing in mind that we want to be "amenable to detailed computerized simulation", it is still often convenient to include conditions and actions at a stage when we are aware of them, but not yet prepared to describe them formally. OOSD encloses these informal descriptions in braces, e.g. *{Initialize Components}* in Figure 6. Although we spoke of broadcast messages and informal descriptions separately in describing actions just now, actually broadcast messages are by their nature informal descriptions, and are thus enclosed in braces.

In Figure 7, the action *Sensor_Fault_Light.Turn_On* in the behavior *Recording* is a request for an operation. The action *{Broadcast Enable(Sensor_Name)}* on the transition from *On.Command_Processing.Wait* to *On.Command_Processing.Wait_for_Acknowledge* is a broadcast message (in this instance received by *Sensor_Management.Testing*). The action *{Recording Status of All Sensors}* in the *On.Recording.Recording_Status* state is an informal description.

When a description of work is associated with a state that may later be expanded, the description summarizes the work refined later in the orthogonal components or substates. For example, the state *On.Recording.Recording_Status* could be expanded to show the detailed behavior represented by the description *{Recording Status of All Sensors}*.[7]

We may wish to abstract away whatever time some particular work may require. We represent such work on a transition. In Figure 5 we have placed *{Initialize.Components}* on the transition between the *Off* and *On* states; relative to the time we expect the system to be *On*, we consider that initializing components takes zero time. When we consider work as taking nonzero time, we associate the work with a state. With these two notations for representing work, which show the abstraction of time in our behavior description explicitly, the object we are examining is always in a defined state at any instant in time, not "in transition".

Note that, more specifically, we have treated initializing components in Figure 6 as taking zero time *for purposes of the current abstraction.* If we later want to expand the transition between *Off* and *On* to show an intermediate state where the work of initializing components is performed, we can treat initializing components as taking substantial time for purposes of that level of abstraction.

OOSD uses the specialized *operation state* and *operation request state* to show the precise timing of synchronous communications (including "timed" and "balking").[8] Timing issues are not critical to our present problem, so these states are only noted for completeness.

---

6 OOSD does not preserve Harel's distinction between work in states, which he calls *activity*, and work on transitions, which he calls *action*.

7 In Harel's nomenclature, work associated with the parent is "concurrent with" the work associated with the substates, but OOSD omits this, because his orthogonal components already show concurrency.

8 In an *operation state*, represented by a hexagon, an object is performing the work immediately associated with one of its operations. The operation state is given the name of the operation (with the operation's formal parameter list, if appropriate). The actions (or substates) of an *operation state* describe the work to performed when the operation is requested. The operation state is shorthand for the frequently occuring pattern *state_name / {perform operation}* which is then expanded to *state_name / {do action_1}, {do action_2},...* where *action_1, action_2,…* describes *perform operation* in detail. While the performing object is in the operation state, the requesting object waits (assuming a synchronous communication); once the performing object leaves the operation state, the requesting object is released.

In many instances OOSD will allow two or more legal representations of behavior. The user should consider whether a particular representation is more suitable for the purpose at hand. Some representations will be equivalent.

Behavior of the System Object

OOSD describes the behavior of the system object as well as the behavior of system components. This promotes uniformity, and guards against un–inspected assumptions about the design. In requirements analysis, we describe the behavior of the system object expected (required) by the customer independent of the components that will be used to build it. For example, Figures 5 and 6 describe the behavior of the *Temperature_Monitor_System*.

We study the described system behavior, in whatever oral or written statements we are given, for the interactions between the system object and external objects (how the system should respond to requests for each of its operations, when and under what conditions it issues requests to external objects), and what other work the system is expected to do. We want a complete and consistent description of the way the system object is expected to behave (see "Product of Requirements" section below).

As a general rule, I recommend starting the behavior description either with *On* and *Off* states, as in Figure 5, or with an *Alive* state, which is the start state, and a termination state (i.e. *Dead*), and then elaborating the *On* or *Alive* behavior. Starting the behavior description at the *On/Off* (*Alive/Dead*) level protects the developer from overlooking the requirements for start–up and shut–down. How is the system created[9] or activated? What initialization must be performed? How is the system destroyed or de–activated, and what "clean up" must be done? Once the system is created, can it be re–activated after it is de–activated?

Our problem statement did not describe the start up or shut down behavior of the Temperature_Monitor_System, so for completeness we proposed (Figure 5) that the Temperature_Monitor_System is required to accept a *Toggle_Power* operation request to switch from *Off* to *On* and vice versa, that the Temperature_Monitor_System will *initialize components* when going *On*, and that some clean–up will be part of the transition to *Off*. We documented our questions on these points in the Dictionary.

Once we have established the expected start up and shut down, we next want to describe the system object when it is *On* (*Alive*). If it is to perform multiple simultaneous behavior, we need to represent the *On* (*Alive*) state in corresponding orthogonal parts, and describe each.

As described in our problem statement, the *Temperature_Monitor_System* must "continually monitor temperatures", and post an alarm if "any of the enabled sensors register an out–of–limits value." It must "record the status of the sensors every fifteen minutes (set by a timer hardware interrupt)." The system must also process user commands. In Figure 6, we represented this simultaneous behavior by expanding the *On* state into the three orthogonal parts *Sensor_Management*, *Recording*, and *Command_Processing*

In an *operation request state*, represented by an oval and named with the operation requested (i.e., *object_name.operation_name* with actual parameters, if appropriate), an object requests that another object perform one of its operations. The operation request state is shorthand for the frequently occurring pattern *state_name / object_name.operation_name (parameters...)*. The object in this state waits for the named object to accept and perform the specified operation. There should be a default transition from the operation request state describing what state the requesting object will change to when released by the performing object. If the *operation request* is timed, there should be a transition with a *time–out* to define how long the requesting object will wait for the performing object to accept the request, and what state the requesting object will change to if the time limit expires.

9 OOSD allows a distinction between creation or destruction on the one hand, which strictly speaking are performed by an external object, and activation or de–activation on the other hand, which are performed by the system object.

respectively. Since the problem description states that there are 10 sensors, we have shown multiple occurrences of the *Sensor_Management* behavior. We have further shown *Value_Monitoring* and *Testing* as orthogonal parts of *Sensor_Management*.

In Figure 6, the orthogonal part *Recording* indicates that the *Temperature_Monitor_System* must periodically record the status of all sensors. *Recording* shows that the system starts in a *Wait* state, and stays there until receiving a request for the *Mark_Time* operation. Upon receiving that request, the system changes to the state *Recording_Status*, and performs the action *{Record Status of All Sensors}*. Upon completing this action, the system will return to *Wait*. If the system *Failed to Connect to Any Sensor*, it issues the request *Turn_On* to the *Sensor_Fault_Light*. We defer until later the details of how the *Temperature_Monitor_System* performs *{Record Status of All Sensors}*, which is thus shown in brackets; we may for now treat this as a design decision.

The behavior diagram and object–interaction diagram (OID) belong to an integrated description of the system. The behavior diagram is not a separate model, but part of the model of the system object. Every interaction shown in the OID is reflected in the behavior diagram, and vice versa. If the OID shows the system object accepting a request to perform an operation, then the behavior diagram must show at least one condition for state change, with the acceptance of the requested operation, and what the system does in response. If the OID shows the system object issuing a request to an external object, then the behavior diagram must show at least one action where the request is issued. Note that if the object (or its class) has been re–used from another system, the behavior diagram may show the acceptance of a request that is not shown as a request in the OID.

Comparing Figure 6 with its context OID (Figure 1), we can see that there is at least one condition for state change for each request of the *Temperature_Monitor_System* shown. We can also see that for each request issued by *Temperature_Monitor_System* as shown in its context OID, there is at least one action where the request is issued. The behavior diagram shows that the *Temperature_Monitor_System* must accept a request for its *Toggle_Power* operation in order to change between its *On* and *Off* states — which we just noticed the need for, and added; there is yet no entry in the context–level OID for the *Temperature_Monitor_System* showing the operation requested, so we must be sure to add this operation here too, with its question marks.

Property Specification

The Object–Interaction Diagrams and the Statecharts show *which objects interact* and *under what conditions*, but not *how well*. For example, Figures 1 and 5–6 do not show how frequently the *Temperature_Monitor_System* can (or is required to) accept the request for its *Write* operation, nor do they show how easy the system is to use. These "how well" issues belong to Property Specification.

The Property Specification activity defines the quality of an object in terms of *operational characteristics* ("properties"). During requirements analysis, we specify the required quality goals and resource constraints for the system object.[10]

Property Forms & Tables

Two textual representations are used: the Property Specification Form and the Property Summary Table (Gilb 1988). The Property Specification Form (see Figure 7) states the unit of measurement ("scale") for each property of an object (including the system object), how the property will be tested ("test"), the minimum ("worst acceptable") and desired ("plan") measures to be met, known best ("record") and past–performance ("past") measures, current ("now") measure, notes, references, and questions. The "test" field should include any separate tests for different phases of the development life–cycle, or for different

10 In the past called the "non–functional requirements", perhaps an unfortunate name.

conditions that may affect the object (e.g. wet roads affecting a car's braking distance); likewise, "worst acceptable", "plan", "record", "past", and "now" should include any special measures that depend on particular conditions. If in filling out this form, or revising it in the process of development, we find a "worst acceptable" which is greater than the "record" or "plan", we thus discover a risk that a property will fail its minimal or desired achievement. Likewise, as we proceed we can watch the achievement of a property by verifying from time to time that the "now" value is in the range defined by "worst acceptable" and "plan".

Figure 7: Property Specification Form for *Write* Frequency

Name:	Write Frequency
Applies to:	Temperature_Monitor_System.Write
Description:	How often the Temperature_Monitor_System.Write operation can be processed by the Temperature_Monitor_System without loss of data.
Scale:	Cycles/Second
Test:	The operation will be initiated repeatedly for 1 hour period at a fixed number of cycles/second, starting at the plan level, decreasing by 1 cycle / second, until each interaction during the hour succeeds. The interval at which all interactions succeed, defines the value for this operation.
Worst Acceptable:	50 cycles/second
Plan:	70 cycles/second
Record:	
Past:	
Now:	
Note:	*Sample Numbers.* The plan level of average interaction is designed as a 40% safety margin (e.g. hardware change).
References:	
Questions	

The Property Summary Table (Figure 8), which summarizes all properties of an object (including the system object), helps analyze trade–offs of properties, risks of failing to achieve desirable or necessary combinations, and alternate designs (see "Product of Requirements" section).

Figure 8: Property Summary Table for *Temperature_Monitor_System*

Name	Applies to	Scale	Worst Acceptable	Plan	Record	Past	Now
Write Frequency	Temperature_ Monitor_ System.Write	cycles/ second	50	70	?	?	
Ease–of–Use. Time–to–Learn	Temperature_ Monitor_ System	weeks	2 (*qualified*)	1 (*qualified*)	?	?	
Ease–of–Use. Productivity	Temperature_ Monitor_ System	operations / hour	30	60	?	?	
Ease–of–Use. Error–Rate	Temperature_ Monitor_ System	errors / hour	3	1	?	?	

Defining the System Object's Properties

Formally defining the properties of an object (including its operations and interactions) is a characteristic feature of OOSD absent in most methods. In requirements analysis, we define the properties required of the system object. We look for quality goals such as those related to reliability, performance, useability, availability, safety, adaptability, security, and for constraints on resources such as equipment, time, people, and money. For each property identified, we fill out a Property Specification form (Figure 7).

Often properties are incompletely described in the documentation we are given. If only a value is specified, we must verify whether the customer meant it as a minimum ("worst acceptable") or a desired value. It is also vital to define, and get our customer's agreement to, a test for the property that will be used to determine whether we have achieved an acceptable level; by making sure of this now, we guard against delivering a product later that is acceptable according to our tests, but a failure according to the customer's. We may have to research past projects, including literature, to determine "past" and "record" values. (Note that we did not attempt to research these for the properties defined in Figures 7 and 8).

We noted above that problem statements often prove to contain design decisions expressed as requirements. We may likewise expect to find some properties stated to us in terms of expected solutions which the customer hopes will achieve a desired but unspecified result. For example, by way of describing *ease of use*, a property, the customer may specify *the currently popular human interface*, a solution ("I need a mouse–windowing environment"). However, specifying a solution does not insure that the desired result will be achieved (Gilb 1988). It is part of our task to recognize when a statement has in fact specified a solution. This usually indicates that the intended property was difficult to define. We attack a hard–to–define property by identifying and defining the factors ("sub–properties") that compose it. For example, productivity, error rate, and time–to–learn are factors in ease–of–use (Figure 8).

To get a complete inventory of the properties a system needs, we must often review similar systems to identify properties omitted from the documentation we are given. While some argue that "if it's not described in the documentation, it's not required", we would be remiss in our duties as hired experts, and as engineers, if we did not pursue our own technical examination of what was needed, and take up any apparent omissions or discrepancies with the customer. Furthermore, if we fail to do so, we run the risk of developing a system that the customer cannot use, and will not want (or if far enough from his needs, may be unable) to pay for.

In developing a complete list of the properties, we may be overwhelmed if we try to achieve them all at once. One strategy for managing this risk is assigning priorities to each of the properties and using an evolutionary process to develop the system (instead of the "waterfall" process which this paper follows for convenience). In the first "build" of the system, we focus on the "critical" top 10 to 20 properties, and in later builds, work on the rest. (Gilb 1988) Such procedures also help evaluate alternate designs.

Object–Class Specification

The Object–Interaction Diagrams, Statecharts, and Property Forms and Tables show *which objects interact*, *under what conditions*, and *how well*, but not *what classes are used to define the objects or the flows*, *the relations between classes*, nor *structure of the classes*. For example, Figures 1 and 5–8 do not show the class of the *Value* data flow that passes from *Physical_Sensor* to the *Temperature_Monitor_System* with the *Write* request. Defining classes is the province of Object–Class Specification.

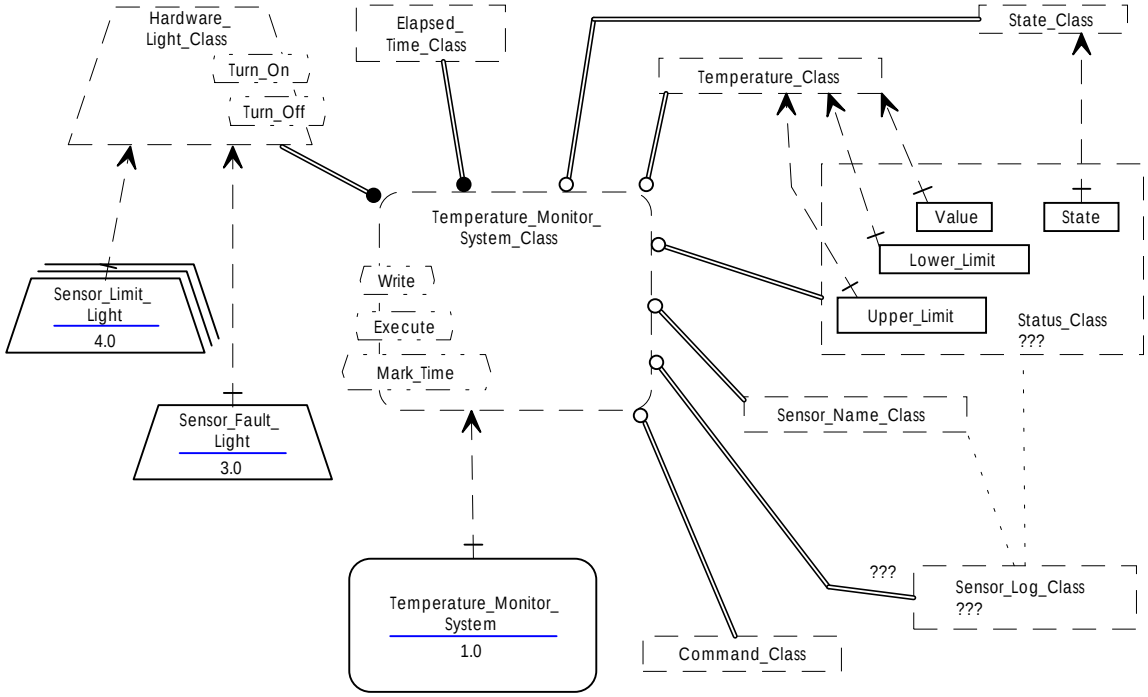
The Object–Class Specification activity identifies and defines the class (including its attributes and structure) of each object, and the relations between the classes, using Object–Class Diagrams and the Dictionary. In requirements analysis, Object–Class Specification defines the classes, attributes, and relations required to describe the system object and its interactions with external objects.

Object–Class Diagram Notation

Object–Class Diagrams (see Figure 9) 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.

An OCD uses the same notation as an OID to describe objects (compare Figure 1). Active objects are shown by round–cornered rectangles (e.g. *Temperature_Monitor_System*), passive objects are shown by square–cornered rectangles (e.g. *State*), and external objects are shown by trapezoids (e.g. *Sensor_Fault_Light*). "Shadowing" (e.g. *Sensor_Limit_Light*) indicates multiple objects of the same class which are distinguished by some form of mapping, e.g. an address. Operations are represented by hexagons (e.g. *Write*) and are shown attached to the symbol for the class that defines the operation. Interface operations, i.e. operations proved by or requested by externals, are shown by lines with a dash–dot–dot pattern. The OCD notation for classes is the same as for objects, except that dashed lines are used (e.g. the external *Hardware_Light_Class*). The names of classes should be nouns with the suffix "_Class"[11].

Figure 9: Object–Class Diagram for the *Temperature_Monitor_System*



An OCD represents nine fundamental relations. Three, "has part" (and its converse "is part of"), "collection of" (and "is member of"), "contains" (and "is contained in"), are represented by graphical enclosure (e.g., *Status_Class* has the parts *Value*, *Lower_Limit*, *Upper_Limit*, and *State*; for "collection of" see Figure 13; "contains" is not used in this paper). The representations for the other six (Figure 10) are adapted from Booch's Class Diagrams (Booch 1991). For example, Figure 9 shows that *Temperature_Monitor_System* and *Sensor_Fault_Light* are instances of the *Temperature_Monitor_System_Class* and the *Hardware_Light_Class* respectively, and that *Temperature_Monitor_System_Class* uses the *Hard-*

11 Since the names of both objects and classes will be nouns, OOSD uniformly includes the suffix "_Class" in the names of classes.

ware_Light_Class and the *Elapsed_Time_Class* in its implementation only, and the other classes in its interface.

An attribute is described with its name, its class, and (if appropriate and known) its value. To avoid cluttering the OCD, the attributes of objects and classes, and the parameters of operations, are described in the Dictionary. For "atomic" classes and objects (i.e. that cannot be further decomposed, or that the current problem will not require us to decompose further), e.g. *Temperature_Class* (Figure 11), OOSD supplies pre–defined attributes, whose values the developer then specifies. The attributes of "non–atomic" classes and objects, e.g. *Temperature_Monitor_System* and *Temperature_Monitor_System_Class* (Figure 11), are determined by inspecting the behavior description.

Figure 10: OCD Relations

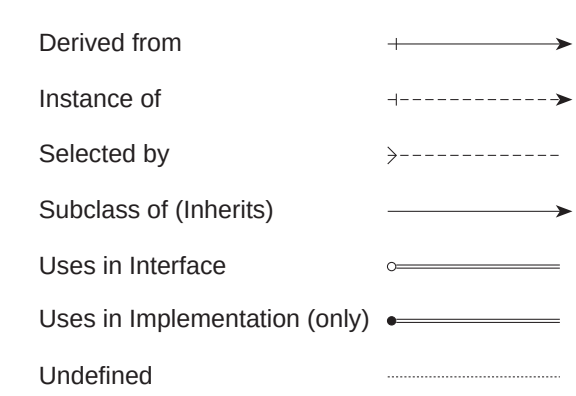


Figure 11: Sample Attribute Descriptions of Classes and Objects

Class/Object	Attributes
Temperature_Class	Values: –273.15 .. 6,049.0 °C (?) Accuracy: 0.5 Physical Representation: 16 bit integer multiple of accuracy
Temperature_Monitor_System_Class	Connection_Period Elapsed_Time Sensor_Test_Period (?) Elapsed_Time …
Temperature_Monitor_System	Connection_Period Elapsed_Time = 5 seconds Sensor_Test_Period (?) Elapsed_Time = ? …

Parameters are described with their name, direction of flow, and class name. Figure 12 shows the parameter descriptions for the operations *Write* and *Execute* of the *Temperature_Monitor_System_Class*.

Figure 12: Sample Descriptions of Operation Parameters

Operation	Parameters		
Temperature_Monitor_System_Class. Write	Value	in	Temperature_Class
Temperature_Monitor_System_Class. Execute	Command	in	Command_Class
	Lower_Limit	in	Temperature_Class
	Upper_Limit	in	Temperature_Class
	Sensor_Name	in	Sensor_Name_Class
	Sensor_Log (?)	out	Sensor_Log_Class

Identifying and Defining Classes

Unlike most object–oriented methods which identify classes by searching for all possible classes, then eliminating or consolidating classes based on method–specific rules, OOSD identifies specific classes by generalizing from the objects in the model, including their attributes and their operation parameters (except when a class is re–used). During requirements analysis, only those classes and relations are described which are actually required to define the external view of the system object. For example, each class in Figure 9 is needed to define the *Temperature_Monitor_System*, its operations, its interactions with other objects, and its attributes, as described in Figures 1, 5, and 6.

We start the Object–Class Specification activity in requirements analysis by identifying the classes of the system object (e.g. *Temperature_Monitor_System*), and all external objects from which the system object requests operations (e.g. *Sensor_Limit_Light* and *Sensor_Fault_Light*).[12] We add a description of each class to the OCD and Dictionary if the class is not yet defined; if we are re–using a class defined earlier in the development, we refine the existing entries as needed; if we are re–using a class from a previous project, we incorporate the definition. We draw an arc for "use in implementation" from the class of the system to each external class.

We add, in the appropriate class, a definition for each operation requested in the OID (e.g. *Write* and *Turn_On*) and not already in the OCD and the Dictionary, including definitions for parameters of the objects and information (e.g. *Value*) exchanged during the interaction, improving or creating subclasses of re–used definitions as appropriate. We identify the class of each parameter, and confirm that an appropriate description is in the OCD and Dictionary. We add or refine attribute descriptions, for the system class, of information (e.g. *Connect_Period*) that must be known by the system, as shown in the system's behavior description. If a new class is used to define an attribute or an operation parameter, we draw a "use in interface" arc from our system class to the new class; otherwise, we draw a "use in implementation" arc.

We create a separate OCD for each new class identified, showing its structure, operations, and relations — easily re–used in other systems. (In Figure 9 we included the detailed view of *Status_Class* to illustrate some of the points in this paragraph.) For each class added to the OCD, we define its required attributes and operations. We study the behavior diagram of the system object to identify the required operations that each class must provide; e.g. the *Temperature_Class* needs to provide comparison operations which our system will use. If the class is atomic, we fill in the values for the predefined attributes (see *Temperature_Class* in Figure 11); if non–atomic, we need to determine what information is known about the class by reviewing the behavior diagram and problem description. For example, our problem description tells us to record the status of a sensor, but omits to specify "status"; from the expected behavior of the system, we concluded that the *Status_Class* should have four attributes, *Value, State, Lower_Limit*, and *Upper_Limit*, and we documented our questions. We need to define the physical representation of any class that is used in an interface, to make sure our system can integrate with its external objects. For example, in Figure 11, we define the physical representation of *Temperature_Class* as "16–bit integer multiple of the accuracy", and in Figure 9, we show that the four attributes of *Status_Class* are implemented as component objects (in the Dictionary entry we would specify bit alignment, etc.). We complete the description of each class by adding appropriate relations between the class and other classes.

---

12 We need not represent the classes of external objects from which our system object requests no operations, because our system does not need to know about them — they need to know about our system. For the convenience of reviewers, the OID includes objects which know about our system, but are not known by our system, by providing a complete picture of the interactions between our system object and all externals.

Validation of the Requirements

We want to verify that our model will meet the customer's needs. We need to eliminate any misinterpretations of the requirements as specified by the customer, and differences between what the customer says and what the customer may actually wish. This is "validation". Although the topic has for convenience been deferred to the end of this paper, if one defers the activity of validation to the end of development one risks substantial inefficiency. OOSD facilitates incorporating validation in the process of development. The following four steps are done from time to time as the activities described above go on.

1) Verify that the various diagrams, forms, and Dictionary that describe the model are consistent. For example, each operation requested of the system object, as shown in the OID, must appear in the Statechart as a condition for some change of state, with the possible responses; and each request issued by the system, as shown in the OID, must appear in the Statechart as some action with the conditions that lead to the request. These consistency checks are necessary, but not sufficient, to establish that the model is complete and correct.

2) Demonstrate that the behavior of the system object is correct. Since the model is a visual prototype of the system, we can demonstrate that the behavior of the system is correct by studying the interactions described in the OID, devising test scenarios, and "executing" the model against the scenarios, automated by support tools as may be available. If the behavior is not what the customer needs, we revise the model and re–test.

3) Verify that the properties as described meet the customer's needs. We review with the customer any properties that pose high risk, i.e. a "worst acceptable" that exceeds the "record" or "plan", confirm that the customer is aware of the risk, and either establish that the risk will be accepted, or explore alternatives. We also need to analyze trade–offs of properties, to discover and see if we can avoid risks of failing to achieve desirable or necessary combinations. We need to identify any properties that conflict, e.g. high efficiency if achieved may lower re–useability. We need to include an evaluation of whether we can achieve acceptable levels for all properties combined, including development cost, schedule, and resources.

4) Verify that we can expect the described behavior on schedule. In our problem, the *Temperature_ Monitor_System* must finish recording the status of sensors before the next *Mark_Time* request arrives[13]; or to consider a hypothetical different problem, a bank might have to process all transactions in time to report daily balances to government supervisory agencies. To analyze whether the system will meet its schedule, I recommend the Rate Monotonic Analysis developed by the Software Engineering Institute of Carnegie Mellon University (Sha 1988; Software Engineering Institute 1990), which predicts mathematically based on the CPU utilization of periodic and aperiodic activities.

Product of Requirements Analysis

By definition, requirements analysis ends when the required context, behavior, and properties of a problem have been clearly identified and validated ("what"). The next step beyond enters preliminary design ("how"). However, managers or customers often extend "requirements analysis" to include the top–level decomposition of the system, so they can review documentation of this level before authorizing the rest of the design. This request can easily be accommodated by supplying the internal view of the system object, developed at the start of preliminary design, along with the results of OOSD requirements analysis.

---

13 The *Temperature_Monitor_System*, which is watching only an office building, has no close timing constraints because it . However, it could as easily be watching temperatures in a jet engine, or a nuclear reactor.

OVERVIEW OF OBJECT–ORIENTED DESIGN

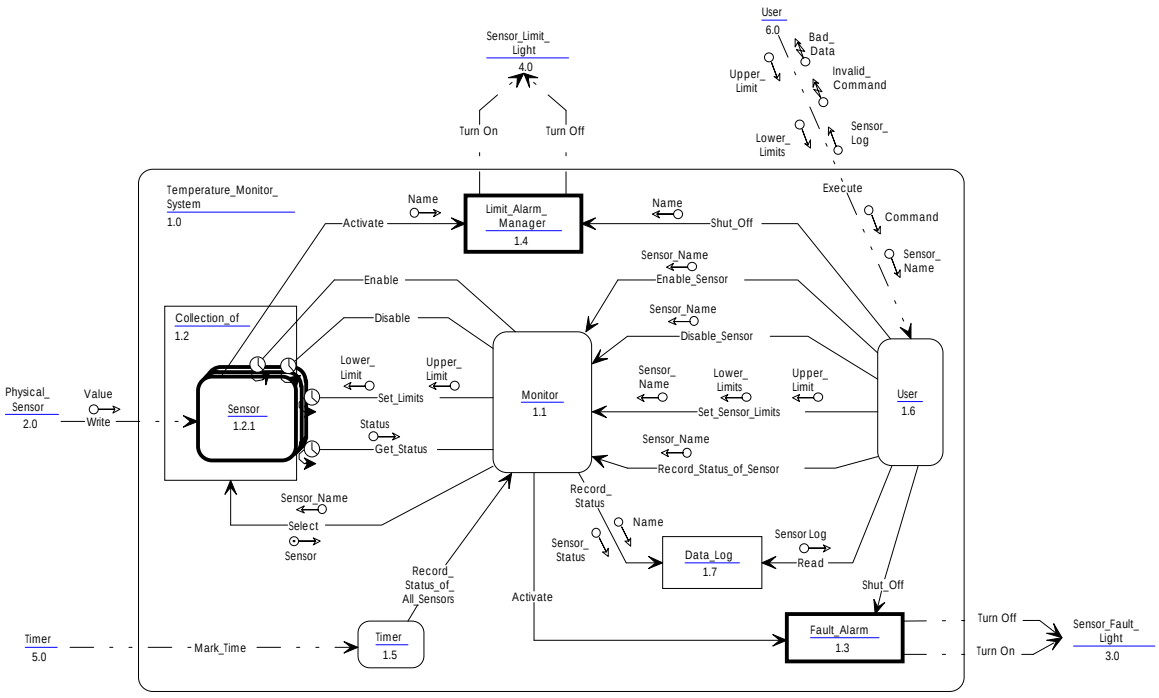
The OOSD approach to design refines the model produced during requirements analysis into a software architecture ("preliminary design"), and creates a representation of that architecture coupled to the implementation language ("detailed design"). The result of preliminary design is a description independent of language and technology (e.g. what kind of network). Detailed design yields a description specific enough to produce code.

Preliminary Design

In preliminary design, the model of the application which was developed during requirements analysis is now made the top level of the architecture. This architecture is then refined, through considering "how" the model will be implemented (see, for example, the elaboration of Figure 1 shown in Figure 13).[14]

The activities of requirements analysis now take new meaning as they are re–used 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–Class Diagrams, Statecharts, and Property Specification Forms and Tables. In addition we create an Object–Hierarchy Diagram which illustrates the "has part" relations of all objects in the system.[15]

Figure 13: *Temperature_Monitor_System* OID, Internal–View



---

14 Note that some of the points mentioned above as properly deferred to design, such as the user interface, will not be taken up in this paper, which only sketches OOSD design activities.

15 Strictly speaking the the Object–Hierarchy Diagram is redundant, since its information is already in the Object–Interaction Diagram, but I have found that the OHD helps some people visualize the construction of the system.

In Object–Interaction Specification, we revise the context OID of our system object, incorporating any changes resulting from our discussions with the customer in the course of validating our model. We then create an OID which describes the internal structure of our system (Figure 13), called an internal or "exploded" view. An internal–view diagram illustrates which objects are part of a higher–level ("parent") object by showing them as inside the higher–level object. Object–Interaction Specification at this stage also shows how a parent interacts with its components, and how the components interact.

Boldface lines in Figure 13 indicate an object that has an internal view, during requirements–analysis. Off–page connectors represent objects that the parent object interacts with.

> *Localization* and *information hiding* are fundamental notions of object–oriented software engineering. The abstraction of an *object* puts components inside, "hiding" them, while offering a visible set of operations which may be requested. Work in the present system, and any changes, are thus localized and easier to follow. The OID enforces both localization and hiding.

The internal view of an object, and diagrams that represent external views of the object, must consistently show all the interactions between the object (or its components) and objects outside. An operation which during requirements analysis we viewed as being requested by a parent object may now, at this stage, prove in fact to be only conceptual, and actually implemented as a request by one or more component objects. An interface operation requested of the parent may likewise prove to be only conceptual, and implemented as a request on a component.

OOSD develops component objects by applying a set of guidelines ("heuristics"), and referring to the system object's properties and to software–engineering goals, to allocate what each parent object is required to know and to do, e.g. a component may be created to manage each external object. We want components that have well–defined responsibilities and co–operate to implement the parent object. Components may be re–used, e.g. building a new system that incorporates an existing system, or may be instances of re–used classes from other systems or other parts of the present system.

With each component object, we repeat the procedure of Behavior Specification, Property Specification, and Class Specification, which we performed for the system object in requirements analysis. If we re–use a component (or a class), we can re–use its behavior diagrams, property tables and forms, and class diagram. If we re–use a component developed outside OOSD, we create diagrams, tables, and forms as needed.

We verify the design of an object (its internal view) against its external view. The design of a non–atomic object is best verified before decomposing any non–atomic components. A good design will correctly implement the parent object's external behavior and interactions, and will achieve the parent object's required minimum for every property. The best design will optimize property requirements.

OOSD allows the unusually complete further verification of comparing the results of "executing" the model represented by the internal OID and Statechart of an object, and the external Statechart of its components, with the results of "executing" the model represented by the object's external OID and Statechart. We thus obtain a recursive proof, although not a mathematical ("formal") proof, that the system is correctly implemented and achieves the required properties. We also re–apply Rate Monotonic Analysis to verify that the design will perform the described behavior on schedule.

OOSD generally begins with the overall OID and then works down (except where re–using), in a "breadth first" instead of "depth first" strategy. "Breadth first" generally reveals most errors in the external view of an object before we have designed its internal structure. With "depth first", once we found errors in an external–view object, we might have to revise all the lower–level objects used to build it.

Preliminary design formally ends when all non–atomic objects have been represented with internal views (created or re–used). A project may, if desired, safely choose to begin detailed design for some or all components without completing preliminary design. For example, we could start detailed design for the

Temperature_Monitor_System without designing the internal structure of the components *Data_Log, Monitor, or User*. If we do carry preliminary design to a formal conclusion, we may get better re–useability, including ease of re–implementation with different technology. This is the usual trade–off of development cost now for potential benefit later.

Detailed Design

We now complete the parallelism of the OOSD process by revisiting the same activities once again, this final time yielding a description detailed enough to generate code. At this stage our four activities (plus verification) are

1) Language–Specific Software Architecture Specification[16]
2) Object–Class Specification
3) Behavior Specification
4) Property Specification

These activities modify the graphic and textual representations of the model already produced, and produce new representations. As before, after a first draft of Software Architecture Specification, the activities can be performed in almost any order; since the performance of each generally suggests refinements in the others, an iterative approach is recommended.

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.

Ada and C++ are typical of languages current in software development. For Ada, OOSD uses a modified form of Ray Buhr's notation (Buhr 1984; Buhr 1990) in graphic representation, and Ada Program Design Language ("PDL") in textual representation. For C++, OOSD uses a notation developed by Mark V Systems Limited for graphic representation, and a C++ PDL (or "pseudo–code") for textual representation. There is a straightforward mapping from the objects and class defined in preliminary design to the Ada or C++ software components. Generally, the graphic representation is produced first, then the textual.

While partial automation is available for representing the objects as software components, some knowledge of the language is needed for good choices between alternate representations (see Figures 14–17). Generation of Ada and C++ PDL from the respective graphics, and vice versa, has been automated.

Object–Class Specification refines the diagrams produced in preliminary design, capturing any language–specific classes and relations. This is mainly documentation maintenance.

Behavior Specification now represents the dynamic behavior of the language–specific architecture, and demonstrates that the behavior of the architecture implements the behavior specified for the model. The language–specific Structure Diagrams and PDL are elaborated to represent the behavior of the software components. Behavior Specification

1) Represents the external view of the behavior of the corresponding object or class in the PDL as a comment, written from the perspective of the user of the component, attached to the declaration of the component.

2) Represents the internal view of the behavior of the corresponding object or class.

---

16 Some users like to split Software Architecture Specification in two. Their first activity refines the OID's developed during preliminary design to show specific technology (e.g. a particular type of network or bus) that will be used, independent of language issues. The second creates the language–specific representation.

a) Updates the Structure Diagram for each component so to capture as much detail as possible in the notation.

b) Creates a body for each component, and describe the implementation of the component in terms of the internal behavior of the object.

c) For each operation of the object or class, creates a body for the language–specific representation (i.e. a subprogram or entry in Ada, or a member function in C++), and describes language–specific logic for the behavior associated with the operation.

3) Adds descriptions of any language–specific components needed to complete the representation.

Figure 14: Ada Structure Diagram *Temperature_Monitor_System*

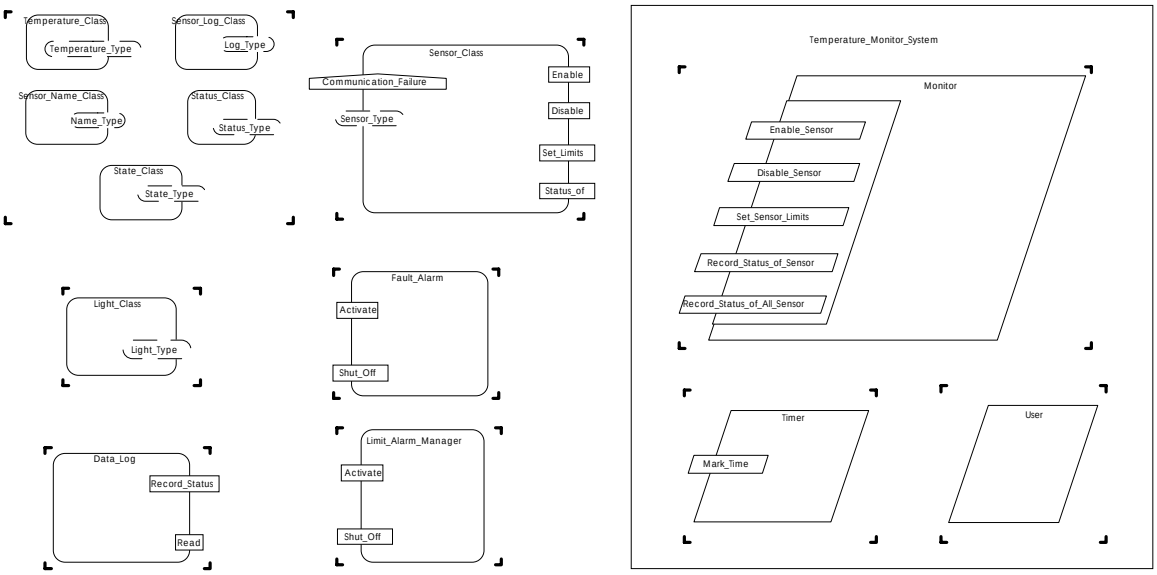


Figure 15: Ada Structure Diagram, *Monitor* Object

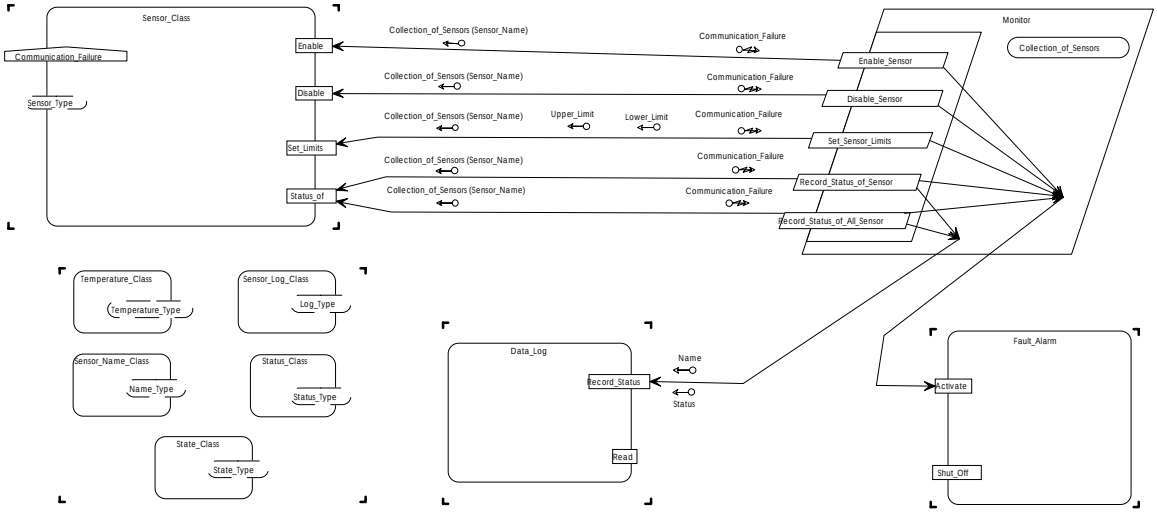


Figure 16: C++ Structure Diagram *Temperature_Monitor_System*

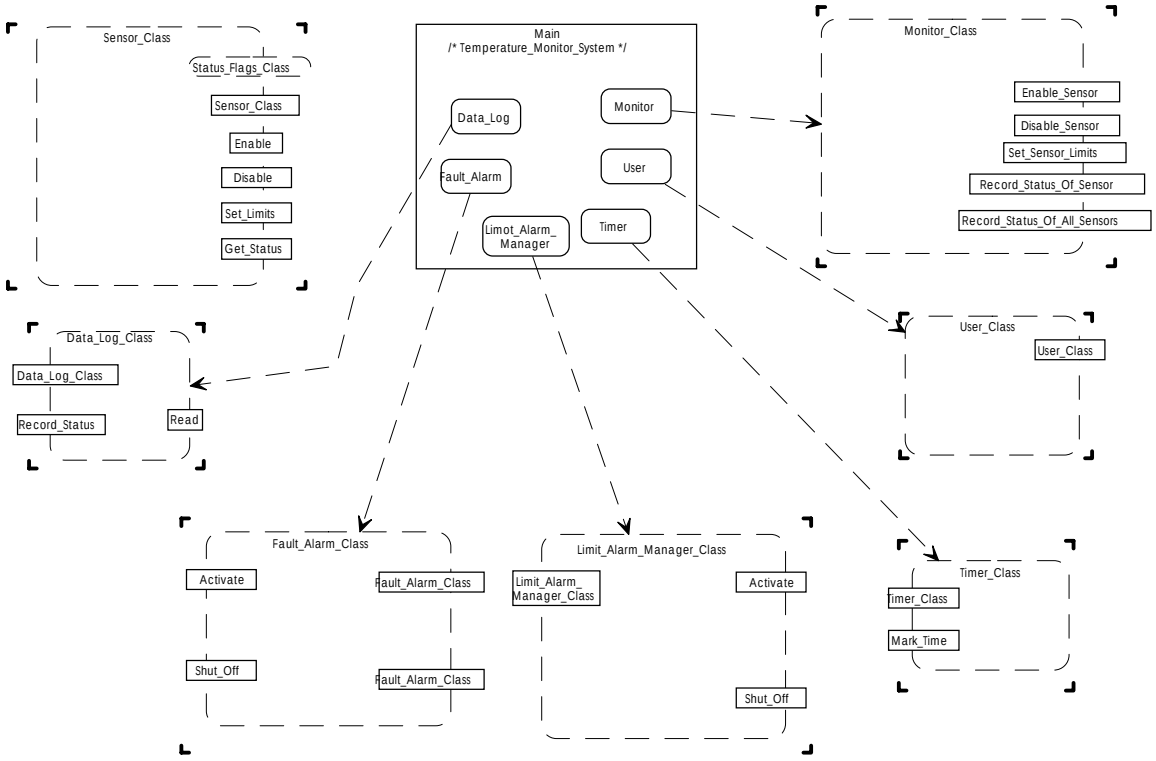
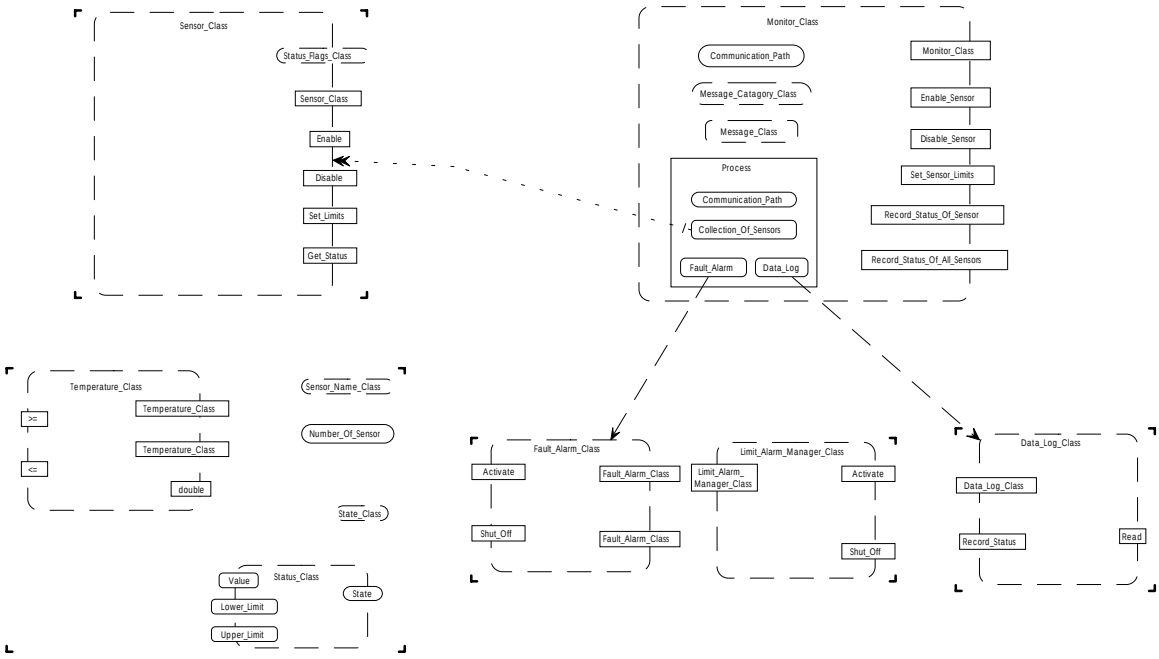


Figure 17: C++ Structure Diagram, *Monitor* Class



Property Specification identifies the properties of each software component, and demonstrates that they meet the required properties of the corresponding objects. The Property Specification Forms and Tables are used to associate the corresponding properties of objects and components, to govern the choices of

mapping to components from preliminary design, and similarly in choosing language–specific implementation.

We verify the language–specific design against the language–independent design. To establish 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, which in turn is done by analyzing or executing 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 PDL). Components must be analyzed or executed to verify that they achieve the required properties. We also re–apply Rate Monotonic Analysis.

Finally, the Dictionary is refined to reflect all work to date as a summary and for cross–reference.

CONCLUSION

OOSD develops a single consistent abstract model of the elements of a problem. During requirements analysis OOSD builds this model from the required objects, classes, functions, behavior, and properties of the problem. During design, the model is refined into an architecture for software components, with a smooth transition to code. The constant refinement facilitates the tracing of requirements throughout the process. Careful study of object interactions and behavior makes this method sensitive to real–time issues.

OOSD allows particularly rigorous behavior analysis. This detects errors early. However, the price is a higher front–end work load, since time and effort must be invested in detailed behavior descriptions (unless descriptions from previous projects can be re–used), and a delay in the production of code (even though code production will be more efficient when it occurs). When there is no human–safety or other substantial reliability concern, a less formal use of the method may be adequate.

OOSD's uniform support for localization and information hiding leads to well–defined objects. Each object is individually testable and provides sufficient information to develop tests. The model developed in requirements analysis provides sufficient information to begin designing integration or acceptance tests for the system, which are then refined as the model is refined. The effects of change can be predicted with assurance, both in the diagrams, forms, and tables of the model, and in the final system.

Formally defining the complete structure, interactions, behavior, and properties of all objects in the system, including the system object, allows validation and verification that the system is correctly implemented. Generating classes from objects in the system yields reliable, re–useable components. Carrying out the same essential activities during requirements analysis, preliminary design, and detailed design, results in unity of the model. The product is easy to understand and maintain, and closely follows the "real world".

REFERENCES

Boehm, B. W. (1981). *Software Engineering Economics*.NJ, Prentice-Hall: Englewood Cliffs.

Boehm, B. W. (1988). "A spiral model of software development and enhancement." *Computer* vol. 21, no. 5 (May), p. 61.

Booch, G. (1986). *Software Engineering with Ada*. Benjamin/Cummings: Menlo Park, CA,

Booch, G. (1991). *Object–Oriented Design with Applications*. Benjamin/Cummings: Menlo Park, CA,

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

Buhr, R. (1990). *Practical Visual Techniques in System Design: With Applications to Ada*. Prentice–Hall: Englewood Cliffs, NJ.

Coad, P. and E. Yourdon (1991). *Object–Oriented Analysis*. Prentice–Hall: Englewood Cliffs, NJ.

Gilb, T. (1988). *Principles of Software Engineering Management*. Addison–Wesley.

Goldberg, A. and D. Robson (1989). *Smalltalk-80, The Language*. Addison-Wesley Series in Computer Science, ed. M. A. Harrison. Addison-Wesley: Reading, MA.

Harel, D. (1987). "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming*. no. 8, p. 231.

Harel, D. (1988). "On Visual Formalisms." *CACM* vol. 31, no. 5 (May), p. 514.

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

Royce, D. W. W. (1970). *Managing the Development of Large Software Systems*. IEEE WESCON, The Institute of Electrical and Electronics Engineers (IEEE).

Rumbaugh, J., M. Blaha, et al. (1991). *Object–Oriented Modeling and Design*. Prentice–Hall: Englewood Cliffs, NJ.

Sha, L. (1988). *An Overview of Real–Time Scheduling Algorithms.* Software Engineering Institute, Departement of Computer Science, Carnegie–Mellon University.

Shlaer, S. and S. J. Mellor (1992). *Object Lifecycles, Modeling the World in States*. Prentice–Hall: Englewood Cliffs, NJ.

Sodhi, J. (1991). *Software Engineering: Methods, Management, and CASE Tools*. Blue Ridge Summit, PA, TAB Professional and Reference Books (division of McGraw-Hill).

Software Engineering Institute (1990). *Rate Monotonic Scheduling Theory: Practical Applications*. Carnegie–Mellon University.

Summerville, I. (1992). *Software Engineering*. Addison-Wesley: Reading, MA.

Wegner, P. (1990). "Concepts and Paradigms of Object-Oriented Programming", *OOPS Messenger*. Vol. 1, No. 1 (August): p. 8.

