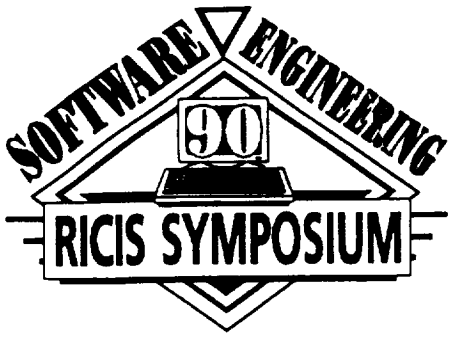


Object-Oriented Requirements Analysis

A Quick Tour

Edward V. Berard

Software Engineering Professional Education Center
University of Houston-Clear Lake
2700 Bay Area Blvd., Box 258
Houston, Texas 77058
(713) 282-2223

OOSE

BERARD

Object-Oriented Approaches to Software Development

Of all the approaches to software development, an object-oriented approach appears to be both the most beneficial and the most popular.

OOSE

BERARD

What Are Objects?

Objects are the physical and conceptual things we find in the world around us. An object may be hardware, software, a concept (e.g., velocity), or even "flesh and blood." Objects are complete entities, e.g., they are not "simply information," or "simply information and actions." (Software objects strive to capture as completely as possible the characteristics of the "real world" objects which they represent.) Finally, objects are "black boxes," i.e., their internal implementations are hidden from the outside, and all interactions with an object take place via a well-defined interface.

BERARD

OOSE

Different Kinds of Objects

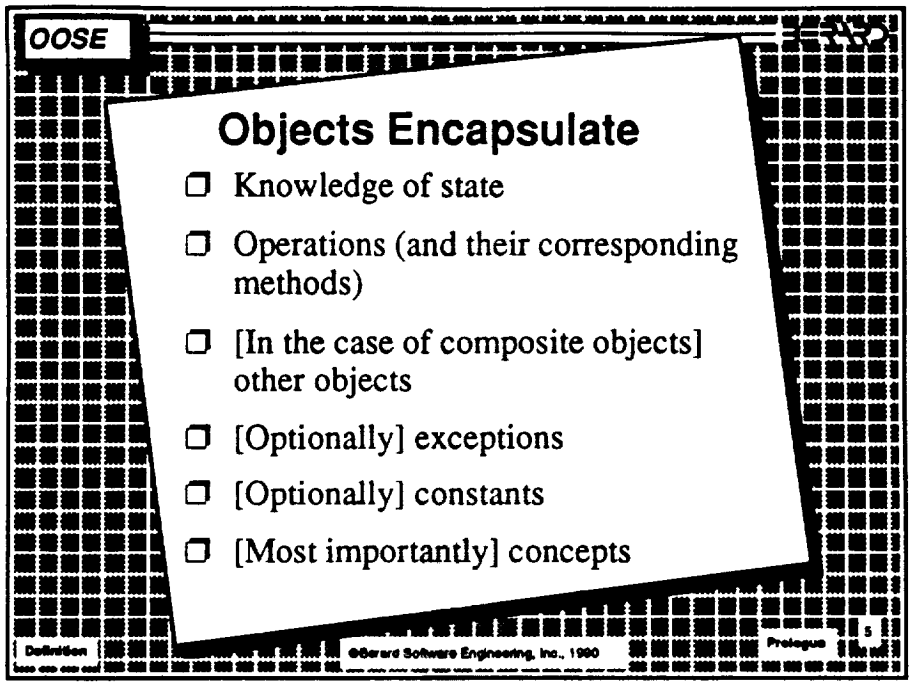
- An **instance** is a specific thing or characteristic.
- A **class** is an object which is used to create instances, i.e., a class is a template, description, pattern, or "blueprint" for a category or collection of very similar items. Among other things, a class describes the interface these items present to the outside world.
- A **value** is the unambiguously defined state (or partial state) of something. A value is an instance of a class, or a collection of instances of (possibly dissimilar) classes.

OOSE

"Truth In Advertising"

While the intention is to keep things simple, there are some things you should know:

- ✔ In object-oriented development in the large (OODIL), we will encounter objects which are not single instances or single classes.
- ✔ Not all "object-oriented" approaches are class-based.
- ✔ Correctly identifying objects is not the most important part of object-oriented software engineering.



OOSE

BERARD

Localization

Localization is the process of placing items in close physical proximity to each other, usually with the connotation of having some mechanism for precisely defining the boundaries of the "area" into which the items have been gathered.

Definition — Prologue — 6

OOSE

BERARD

Different Approaches Mean Different Localizations

- ❒ Functional decomposition approaches localize information around *functions*.
- ❒ Data-driven approaches localize information around *data*.
- ❒ Object-oriented approaches localize information around *objects*.

OOSE

BERARD

What Kinds of Items Will Be Largely Inappropriate For Object-Oriented Software Engineering

- ❒ Data flow diagrams
- ❒ Entity relationship diagrams
- ❒ Relational databases
- ❒ Structure charts
- ❒ Data dictionaries

OOSE

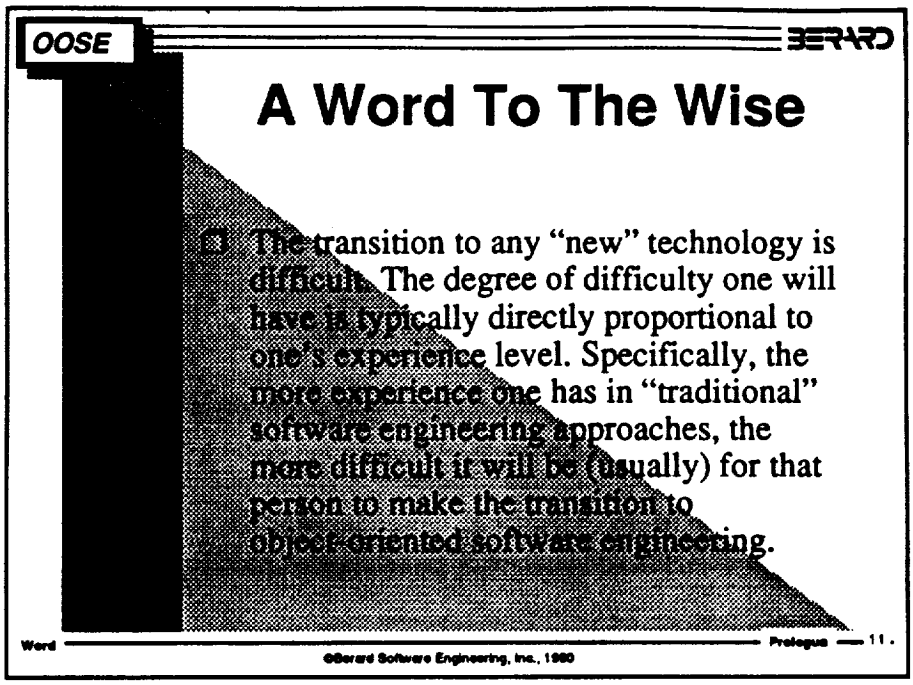
Object-Oriented Software Engineering

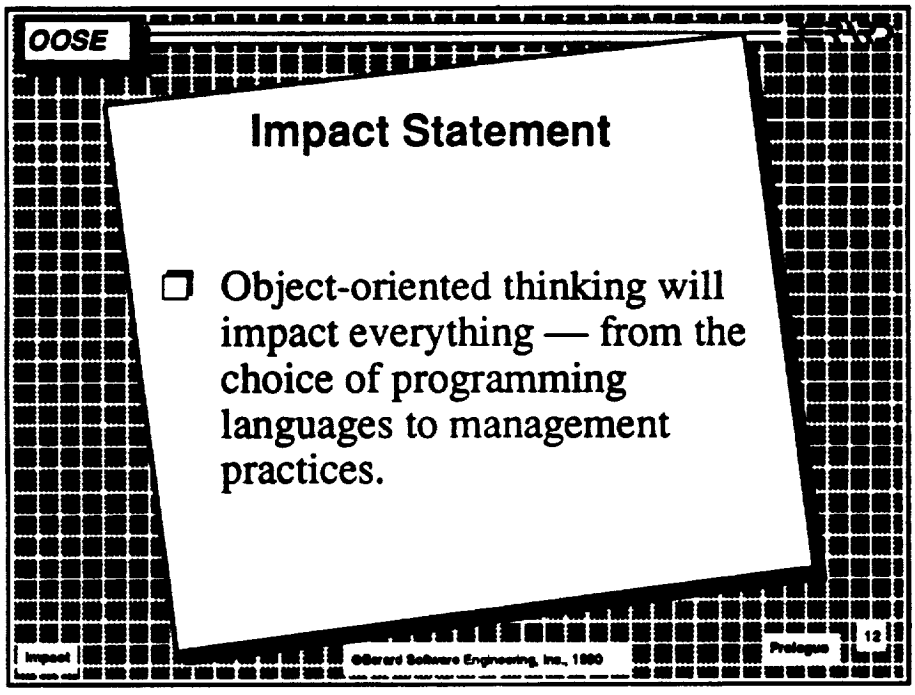
- Identifying relevant objects
- Documenting these objects
- Applying configuration management to the objects of interest
- Producing both static and dynamic object-oriented models of the system.

OOSE

What We Will Find Out About Object-Oriented Software Engineering

- Object-oriented software engineering involves more than just identifying objects of interest.
- Object-oriented development in the large (OODIL) is different from small object-oriented development.
- Object-oriented development of real-time systems brings in its own set of issues.
- Object-oriented software engineering can be quite systematic, and quite formal (i.e., mathematical).





OOSE

BERARD

Motivation for an Object-Oriented Approach

- Motivation for Object-Oriented Approaches in General
- Motivation for an Overall Object-Oriented Approach to Software Engineering

OOSE

BERARD

Motivation for an Object-Oriented Approach In General

- Object-Oriented Approaches Encourage Modern Software Engineering.
- Object-Oriented Approaches Promote Software Reusability.
- Object-Oriented Approaches Facilitate Interoperability.
- Object-Oriented Approaches Closely Resemble the Original Problem.
- Object-Oriented Software Is Easily Modified, Extended, and Maintained.
- General Electric Report

OOSE

BERARD

Object-Oriented Approaches Encourage Modern Software Engineering

- ❒ OOAs require data abstraction
- ❒ OOAs require information hiding
- ❒ OOAs require localization above the subprogram level
- ❒ OOAs naturally support concurrency

OOSE

BERARD

OOAs Promote Software Reusability

[Brown and Quanrud, 1988]
[Ratcliffe, 1987]
[St. Dennis et al, 1986]
[Schmucker, 1986b]
[Russell, 1987]
[Cox, 1986]
[Booch, 1987]
[Tracz, 1987]
[Ledbetter and Cox, 1985]
[Embly and Woodfield, 1987]
[Chan, 1987]
[Margono and Berard, 1987]

OOSE BERARD

OOAs Promote Interoperability

Consider a computer network with different computer hardware and software at each node. Next, instead of viewing each node as a monolithic entity, consider each node to be a collection of (hardware and software) resources. **Interoperability** is the degree to which an application running on one node in the network can make use of a (hardware or software) resource at a different node on the same network.

OOSE BERARD

Interoperability Examples

Consider a network with a CRAY supercomputer at one node rapidly processing a simulation application, and needing to display the results using a high-resolution color monitor. If the software running on the CRAY makes use of a color monitor on a Macintosh IIci at a different node, that is an example of interoperability. Another example would be if the Macintosh IIci made use of a relational DBMS which was resident on a DEC VAX elsewhere on the network.

OOSE BERARD

Polymorphism

In the context of object-oriented languages, association of generic names with behaviors is called overloading of operator names or polymorphism. ... The advantage of this encapsulation is that the programmer need keep track of the names of only a (relatively) few behaviors that are exhibited by a set of objects; the names of the larger set of procedures that implement the behaviors need not be remembered ... [I]t is sufficient for a programmer to know the name of an abstract behavior to invoke it.

—Smith, Barth and Young in [Shriver and Wegner, 1987]

OOSE BERARD

An Example of Polymorphism

Consider the Apple Macintosh. On the screen (desktop) of a Macintosh, there might be several icons, one representing a document, one representing an application, and another representing an uninstalled desk accessory. Sending the same message (for example "duplicate") to each of these objects will cause each to behave in the same general manner (for example, each will make a copy of itself).

OOSE

BERARD

A Better Definition of Polymorphism

Polymorphism is a measure of the degree of difference in how each item in a specified collection of items must be treated at a given level of abstraction. Polymorphism is increased when any unnecessary differences, at any level of abstraction, within a collection of items are eliminated.

OOSE

BERARD

OOAs Facilitate Interoperability

Rather than attempting to localize based on functionality, but instead localizing on objects facilitates interoperability. This is because viewing the system components in object-oriented terms encourages software engineers to describe the same general object behavior using the same names. (The X Windows System, for example, is very object-oriented.)

OOSE

BERARD

Object-Oriented Solutions Closely Resemble the Original Problem

The "real world" may be considered to be highly object-oriented. All entities (e.g., cars, people, and computers) may be viewed as being composed of highly independent objects (interchangeable parts). A solution which duplicates this arrangement in software will resemble the original problem.

OOSE

BERARD

OOAs Promote Software Solutions Which Are Easily Modified

The concept of interchangeable parts not only helped spur the Industrial Revolution, but is an object-oriented concept which facilitates modification and extension of modern electronic equipment. The same concept has been applied with success to software systems, e.g., the Apple Macintosh and the X Windows System.

BERARD

OOSE

BERARD

The General Electric Report

Deborah Boehm-Davis and Lyle Ross conducted a study for General Electric comparing several development approaches for Ada software (i.e., Structured Analysis/Structured Design, Object-Oriented Design, and Jackson System Development). They found that the object-oriented solutions, when compared to the other solutions:

- ✔ were simpler (using McCabe's and Halstead's metrics)
- ✔ were smaller (using lines of code as a metric)
- ✔ appeared to be better suited to real-time applications
- ✔ took less time to develop

OOSE

BERARD

The Impact of Object-Orientation on Software Engineering Processes

Originally, people thought of "object-orientation" only in terms of programming languages. Discussions were chiefly limited to "object-oriented programming" (OOP). However, people quickly found that:

- ✔ object-oriented programming alone was insufficient for large and/or critical problems, and
- ✔ object-oriented thinking was largely incompatible with traditional (e.g., functional decomposition) approaches.

OOSE

BERARD

The Progression of Object-Oriented Technology

- ❒ Object-Oriented Programming (1966, 1970)
- ❒ Object-Oriented Design (1980)
- ❒ Object-Oriented Computer Hardware (1980)
- ❒ Object-Oriented Databases (1985)
- ❒ Object-Oriented Requirements Analysis (1986)
- ❒ Object-Oriented Domain Analysis (1988)

OOSE

BERARD

Lessons Learned From Early Projects With OOD

- ❒ Attempting to use object-oriented design with functional requirements is difficult.
- ❒ Localization around functions tends to split high level system objects and increases integration time; whereas an object-oriented front end would identify system objects and maximize the advantage of an object-oriented approach.
- ❒ The benefits of object-oriented approaches are magnified the earlier in the life-cycle one begins thinking in terms of objects.

OOSE

BERARD

Changing Our Thinking On Methodologies

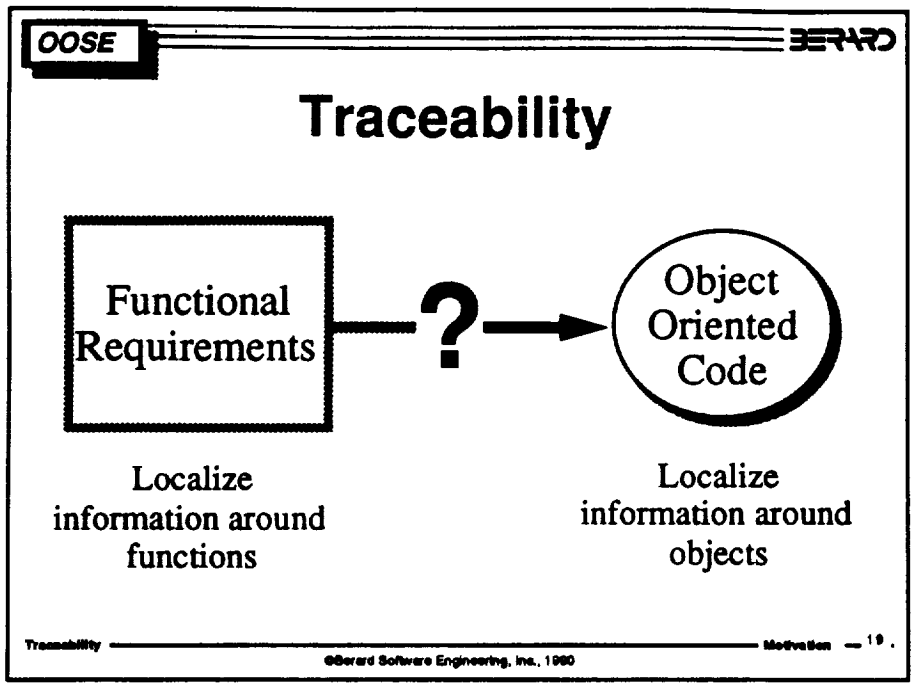
In the 1970s and early 1980s, many people believed that the various "life-cycle phases" (e.g., analysis, design, and coding) were largely independent. Therefore, one supposedly could use very different approaches for each phase with only relatively minor consequences. For example, one could consider using structured analysis with OOD. This line of thinking, however, was found to be largely inaccurate.

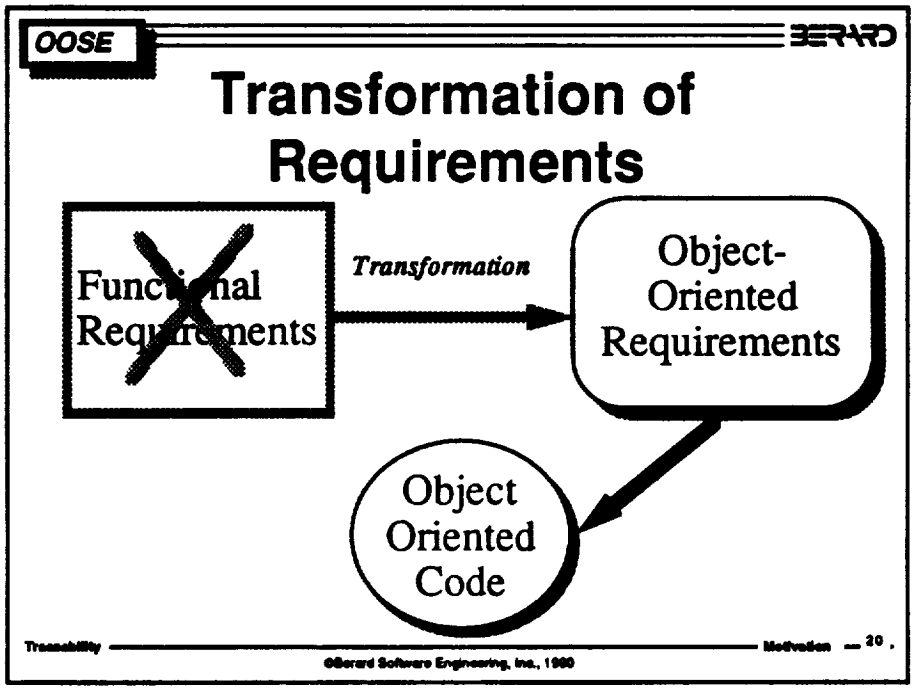
OOSE

BERARD

Motivation For An Overall Object-Oriented Approach to Software Engineering

- ❒ Traceability
- ❒ Reduction of Integration Problems
- ❒ Improved Conceptual Integrity
- ❒ Objectification and Deobjectification





OOSE

BERARD

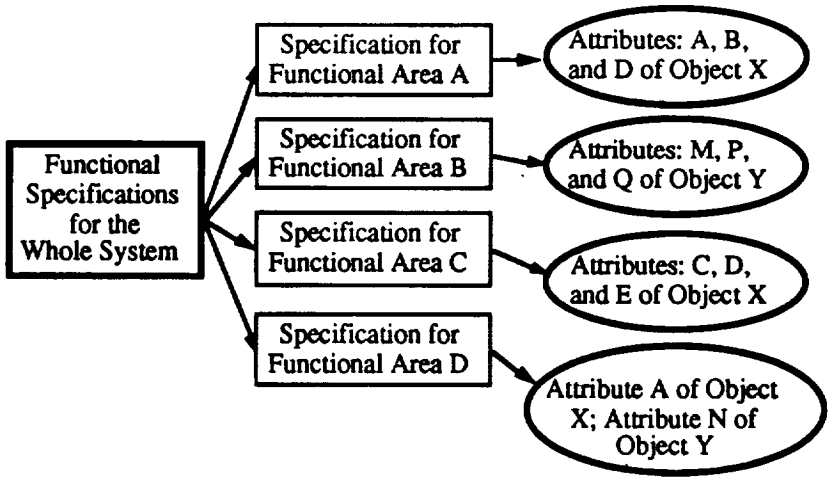
Reduction of Integration Problems

Often, different functional areas of a large project are implemented by different teams of software engineers. If an object is common to several functional areas, but has different attributes in each area, it is likely that each team will implement the same object in a different manner. The serious consequences of this will not become known until the integration phase.

OOSE

BERARD

Different Views of The Same Object



BERARD

OOSE

BERARD

Facilitation of the Introduction of Other Object-Oriented Techniques

Interviews of those who have used object-oriented design on large projects shows that two points are mentioned most frequently:

- ✔ attempting to use object-oriented design with functional requirements is difficult, and
- ✔ the benefits of object-oriented approaches are magnified the earlier in the life-cycle one begins thinking in terms of objects.

OOSE

BERARD

Improved Conceptual Integrity

- ❒ **Conceptual integrity** might be defined as "being true to a concept," or more simply as "being consistent."
- ❒ The more consistent the approach used in the development of software, typically the more reliable, more maintainable, and more usable the software becomes.

OOSE

BERARD

Consistent Solutions

- It has been observed that an object-oriented approach to software development seems to yield results which are superior to those yielded from a functional decomposition approach. Unfortunately, at present, it is all too common to functionally decompose a system at a high level and then attempt to apply an object-oriented approach to the components.
- This approach is hardly consistent.

OOSE

BERARD

Objectification and Deobjectification

Object-oriented approaches and systems localize information around objects. Whenever an object-oriented system must interface with a non-object-oriented (or weakly-object-oriented) system, transformations are usually required:

- ✔ **Deobjectification** is a process whereby an object is decomposed into components which can be understood by the non-object-oriented system.
- ✔ **Objectification** is the process of reconstructing an object from more primitive components. The usual connotation is that these components were supplied by a non- (or weakly-) object-oriented system.

OOSE

BERARD

Examples of When Objectification and Deobjectification Are Necessary

Probably the two most common examples of objectification and deobjectification are:

- ✔ When an object-oriented application requires persistent objects and attempts to store objects in a relational database. To store objects, the objects must first be deobjectified. To retrieve the objects, the objects must be reconstituted (objectified) from the information stored in the relational database.
- ✔ In a distributed application, it will be necessary to transmit objects over the communication links between the nodes. Since few, if any, communications systems are object-oriented, objects will have to be decomposed (deobjectified) before transmission and reconstituted (objectified) upon receipt.

OOSE

BERARD

Minimizing the Need For Objectification and Deobjectification

Since objectification and deobjectification have a negative impact on reliability and efficiency, we wish to avoid them when possible. An overall object-oriented approach to software engineering helps us to minimize the need for these processes.

OORA

Analysis Versus Design

- ❒ Problems With Requirements
- ❒ Analysis Tells "What" — Not "How"
- ❒ Design Tells "How"
- ❒ Defining the Border Between Analysis and Design
  - ✔ The Traditional Meaning of Analysis
  - ✔ Design Tends to be Programming Language Specific
  - ✔ User Visibility
  - ✔ Management Decision Points
  - ✔ Concept of Scope
  - ✔ Concept of Viewpoint
- ❒ Truth In Advertising

OORA

Analysis Vs. Design (Continued)

- ❒ Sources of Confusion
- ❒ Types of Requirements

OORA

BERARD

Problems With Requirements

To gain a better understanding of what is required in analysis, we must first understand that software engineers are seldom presented with a "clean" set of requirements. Often, software engineers will have to improve upon, or create, the initial set of "requirements."

OORA

BERARD

Be Suspicious of the Quality of Any Existing Requirements

Existing requirements usually have one or more of the following problems:

- ✔ omissions
- ✔ contradictions
- ✔ ambiguities
- ✔ duplications
- ✔ inaccuracies
- ✔ too much design
- ✔ irrelevant information

BERARD

OORA

Omissions

- Very often the initial set of user-supplied requirements (and information) is incomplete. This means that, during the course of analysis, the software engineer will have to either locate, or generate, new information. This new information is, of course, subject to the approval of the client.
- Note that this location or generation of new information may be considered by some to be "design."

OORA

Contradictions

- Contradictions may be the result of incomplete information, imprecise specification methods, a misunderstanding, or lack of a consistency check on the requirements.
- If the user alone cannot resolve the contradictions, the software engineer will be required to propose a resolution to each problem.

OORA

Ambiguities

- Ambiguities are often the result of incompletely defined requirements, lack of precision in the specification method, or a conscious decision to leave their resolution to the software engineers performing analysis.
- Resolution of ambiguities may require some " requirements design" decisions on the part of the software engineers.

OORA

Duplications

- Duplications may be the outright replication of information in the same format, or the replication of the same information in several different places and formats. Sometimes duplications are not obvious, e.g., the use of several different terms to describe the same item.
- Software engineers must be careful when identifying and removing duplications.

OORA

BERARD

Inaccuracies

- It is not uncommon for software engineers to uncover information which they suspect is incorrect. These inaccuracies must be brought to the client's attention, and resolved.
- Often, it is not until the client is confronted with a precisely-described proposed solution that many of the inaccuracies in the original requirements come to light.

OORA

BERARD

Too Much Design

One of the greatest temptations in software engineering is "to do the next guy's job," i.e., to both define a problem and to propose a (detailed) solution. One of the most difficult activities during analysis is the separation of "real requirements" from arbitrary (and unnecessary) design decisions made by those supplying the requirements.

OORA

Metarequirements

A "metarequirement" is a stipulation of how a particular system behavior is to be accomplished which is both supplied and required by the client. For example, a client might require that data be encrypted using a specific algorithm. Metarequirements are design decisions made by the client. However, they should be kept to a minimum.

OORA

Failure To Identify Priorities

- A software engineer must have some basis for making decisions. Without a clearly-defined, well thought out, and comprehensive set of priorities, it will be difficult to select from a number of alternatives.
- Software engineers must realize that emphasis on one priority often inversely impacts several others.

OORA

BERARD

Irrelevant Information

Software engineers are often reluctant to throw away any information. Their clients often feel it is better to supply too much information rather than too little. Without some clear cut mechanisms to identify and remove irrelevant information, it will be difficult to develop accurate, cost-effective, and pragmatic solutions to a client's problems.

OORA

BERARD

Analysis Tells "What" — Not "How"

It is the job of analysis to describe *what* is needed, not *how* it is to be accomplished. Software engineers must be aware that there is a very strong tendency to describe *how* while failing to accurately describe *what* needs to be done.

BERARD

OORA

Examples of "What"

- The system must recognize valid Ada source code.
- If a node on the network "goes down," the network will dynamically re-configure itself.
- When the power is turned on the system will conduct a self test, i.e., a power on self test or "post."
- Users will be able to add items to the database, delete items from the database, and be able to determine how many items are currently in the database.
- The product will handle up to 1000 transformations every 10 milliseconds.
- All calculations must be accurate to 10 significant digits.

OORA

Examples of "How"

- An old item may be exchanged for a new item by first deleting the old item, and then adding the new item.
- Trigonometric functions will be evaluated using infinite series approximation.
- Incoming data will be sampled 200 times a second with a statistical analysis done to remove any "system noise," thus assuring the user of "clean data."
- To place a text window on the screen, the user must first create a workstation, then create the text window, and finally, add the text window to the desktop.

OORA

BERARD

Design Relates To "How"

Once the client's requirements have been established, it is the software engineer's job to design a system which will meet these requirements. It is during design that a software engineer must describe the details of *how* the behavior of the system is to be accomplished — within the constraints stipulated by the client.

OORA

BERARD

"What" Vs. "How" Is Insufficient

While saying that analysis should address "what," and not "how" is accurate, it does not provide any detailed guidance to the analyst. We must better define the differences between analysis and design. Further, it is also desirable to define the types of products one expects from analysis.

BERARD

OORA

BERARD

The Border Between Analysis and Design

- The Traditional Meaning of Analysis
- User Visibility
- Design Tends to Be Programming Language Specific
- Management Decision Points
- Concept of Scope
- Concept of Viewpoint

OORA

BERARD

The Dictionary Meaning of Analysis

- The separation of a thing into the parts or elements of which it is composed
- The examination of a thing to determine its parts or elements; also a statement showing the results of such an examination

—Merriam-Webster Dictionary

OORA

BERARD

The Traditional Meaning of Analysis

In technical endeavors, we expect the following things from an analysis effort:

- ✔ an examination of a concept, system, or phenomenon with the intention of accurately understanding and describing that concept, system, or phenomenon,
- ✔ an assessment of the interaction of the concept, system, or phenomenon with its existing or proposed environment,
- ✔ proposal of two to three alternative solutions for the client with an accurate and complete analysis of the alternatives, and
- ✔ an accurate and complete description of the solution to be delivered to the client.

OORA

BERARD

The Traditional Border Between Analysis and Design

An examination of virtually all software requirements analysis methodologies shows that requirements analysis ends with the description of the "user interface." "User" may be taken to mean anything from a human user, to other software, to computer hardware.

OORA

Design Tends To Be Programming Language Specific

- In truth, requirements analysis for software applications is somewhat influenced by the choice of programming language. However, most approaches to requirements analysis strive to be independent of programming language considerations.
- Most of the suggested approaches to software design, on the other hand, deal with programming language concepts (e.g., modules, packages, and software interfaces) directly.

OORA

User Visibility

- "User visibility" is a term used to describe the level of client involvement during the software life-cycle. User visibility is highest during the "analysis" and "use" phases. User visibility is lowest during the "design" and "coding" phases.
- Once the client has accepted the solution described by the analyst, the solution is then turned over to the designers.

OORA

Confidence In the Solution

Both the client and the software analyst must be comfortable with the solution proposed by the analyst. Each must have a high degree of confidence that the resulting system will perform as expected. Further, the designers should have both an accurate description of what they must deliver, and a means of judging the merits of each design alternative they may consider.

OORA

Goals for the Analyst

- The requirements analyst must describe "what the system must do," and avoid the details of "how the system will accomplish its objectives." The analyst must also reduce the need for the user to be "visible" during design.
- Therefore, the analyst must describe the system precisely enough to accomplish these goals.

OORA

BERARD

Management Decision Points

- The end of each phase (or partial phase) of the software life-cycle is a decision point. Management must often make decisions on how to proceed, or whether to proceed. Without a system specified in sufficient detail, meaningful decisions are often difficult, if not impossible.
- The requirements analyst must propose a solution in sufficient detail to allow meaningful management decisions.

OORA

BERARD

The Concept of Scope

In conducting any form of analysis, a software engineer must define the boundaries of the system being analyzed, i.e., the **scope** of the system. There are two generally recognized levels of scope, and, depending on how the software engineer attacks the problem, he or she may deal with only one level.

OORA

BERARD

Two Levels of Scope

There are two instances where there will be two levels of scope:

- ✔ When the analysis must include a model of an existing process which is to be automated.
- ✔ When the analysis includes items other than software, e.g., hardware and users.

OORA

BERARD

Beginning With Two Levels of Scope

In either of the two previously mentioned situations, the software engineer will start with an overall system model, and later focus on the actual software system. The first scope level will include non-software items. The second level will include only the software.

OORA

One Level of Scope

Sometimes, the software engineer will choose to focus solely on the software system during the analysis. In this case, only one level of scope is defined, i.e., the boundary of the software with the rest of the overall system.

OORA

The Concept of Viewpoint

Very often during analysis, software engineers almost automatically pick the viewpoint of a user of the system. Sometimes this results in an awkward analysis and design. Shifting one's viewpoint can result in a simplification of the analysis.

OORA

Truth In Advertising

- The concepts of "life-cycle phases," e.g., analysis, design, and testing, are artificial concepts introduced primarily by management to provide some level of monitoring and control over a software engineering project.
- As we will see later, the object-oriented life-cycle is different. Specifically, it is a recursive/parallel life-cycle, which means that analysis will be performed at many different points, rather than "all at once."

OORA

Sources of Confusion

The following items seem to confuse software engineers:

- ✔ Some items are mentioned (described) in analysis, but do not become part of the design, i.e., they are unique to the analysis.
- ✔ Some items are created in design, i.e., they were not even mentioned during analysis.
- ✔ All that is not expressly forbidden is allowed.
- ✔ All that is not expressly allowed is forbidden.

OORA

Types of Requirements

There are three major types of requirements:

- ✔ User Driven
- ✔ User Reviewed
- ✔ User Independent

OORA

User-Driven Requirements

- ❒ User-driven requirements are requirements which are defined and specified entirely by the client. The software engineers responsible for developing a solution which meets the user-driven requirements have little, or no, input to the definition and specification of the system requirements.
- ❒ This is *not* a desirable situation.

OORA

BERARD

User-Reviewed Requirements

User-reviewed requirements are requirements which are specified by the client and software engineers working together. It is not the software engineers' job to be an expert in the client's application domain. It is, however, required that software engineers possess the skills, methods, techniques, and tools which will enable them to effectively define and specify requirements through interactions with the client.

OORA

BERARD

User-Independent Requirements

User-independent requirements are those requirements which must be defined and specified without a particular user being present. The most common example of user-independent requirements are those requirements which are defined by software product vendors when they choose to develop a new software product.

BERARD

OORA

The Overall OORA Process

- ❒ Understanding What Comes Before OORA
- ❒ Understanding the Mechanisms By Which the Process May Be Accomplished
- ❒ Identifying Sources of Requirements Information
- ❒ Characterizing the Sources of Requirements Information
- ❒ Identification of Candidate Objects
- ❒ Building Object-Oriented Models of Both the Problem and Potential Solutions, As Necessary

OORA

The Overall OORA Process
(Continued)

- ❒ Re-Localization of Information Around the Appropriate Candidate Objects
- ❒ Selection, Creation, and Verification of OCSs, Subsystem Specifications, and Systems of Objects Specifications
- ❒ Assigning OCSs, Subsystems, and Systems of Objects to the Appropriate Section of the OORS
- ❒ Development and Refinement of the Qualifications Section
- ❒ Development and Refinement of the Precise and Concise System Description

OORA

BERARD

Domain Analysis Comes Before OORA

Hopefully, an organization has conducted some form of object-oriented domain analysis. This analysis must:

- ✔ Identify reusable objects within the appropriate application domain(s)
- ✔ Document these objects
- ✔ Place these objects into some form of reusability system

OORA

BERARD

Feasibility Studies May Be Conducted Before OORA

- ❒ One, or more, feasibility studies may be accomplished before OORA for a given project. Feasibility studies often use the techniques of analysis and design, although usually on a much more informal basis.
- ❒ There are several types of feasibility, e.g., financial, technical, time-related, and marketing.

BERARD

OORA

BERARD

Understanding the Mechanisms by Which the OORA Process May be Accomplished

- ❒ People Think Differently
- ❒ Each Project Is Different
- ❒ OORA Is an Iterative Process
- ❒ Some Processes May Be Accomplished Concurrently
- ❒ The OORA Effort May Be Accomplished At More Than One Point In the Life-Cycle
- ❒ The OORA Effort May Be Driven By External Circumstances
- ❒ OORA Should Not Be Driven By Low-Level Issues
- ❒ Verification, Validation, and Software Quality Assurance Are Always Important

OORA

BERARD

People Think Differently

- ❒ It is unusual for two different people to approach the same task in an identical manner — at least on a microscopic level. It is quite normal for individuals to impose their own problem-solving techniques within a specified approach to a life-cycle process.
- ❒ For example, some people may be quite happy to identify objects first, and worry about how they will be used to solve the clients problem later. Others must have an understanding of the overall problem before they can even begin to work.

BERARD

OORA BERARD

Each Project Is Different

There are many things which affect how OORA, or any analysis process, is accomplished. These factors tend to change with each project. For example, a sequential approach may work for a small project, whereas a highly concurrent and iterative approach to OORA may be better suited to a large project.

OORA BERARD

OORA Is an Iterative Process

At a particular level of abstraction, for a particular system of objects, or for a particular subsystem the OORA analyst may make several passes through the process, to introduce changes and/or successively refine the analysis. The larger the project, the greater will be the tendency to make several passes (i.e., iterate).

BERARD

OORA

BERARD

Some Processes May Be Accomplished Concurrently

Although some may attempt to portray OORA as strictly sequential, there are often parts of the process which may be accomplished in parallel. For example, one may simultaneously be identifying objects of interest, and building an object-oriented model of either the problem or a proposed solution.

OORA

BERARD

OORA May Be Accomplished At More Than One Place In the Life-Cycle

The object-oriented life-cycle is a "recursive/parallel" life-cycle. This means that, rather than all the analysis being completed up front (as in a waterfall life-cycle), analysis may be distributed throughout the life-cycle (i.e., "analyze a little, design a little, implement a little, and test a little). What is important is that we perform the analysis which is appropriate to the view of the system at hand.

OORA

BERARD

OORA May Be Driven By External Circumstances

The OORA process may be impacted by such things as:

- ✔ the availability of information
- ✔ the introduction of changes
- ✔ the availability of staff and other resources
- ✔ an increasingly better understanding of the problem

OORA

BERARD

OORA Should Not Be Driven By Low-Level Issues

Except for true metarequirements, OORA decisions should not be driven by low-level issues, e.g., how a concept might be implemented in a given programming language.

OORA

BERARD

Commission Vs. Omission

The two primary life-cycle "sins" are commission and omission. Commission implies that we specify too much detail too soon. Omission implies that we leave out critical information. These two "sins" must be constantly balanced. If we supply too much detail at one point we will restrict our choices at a later point. If we do not supply enough information at one point, we may have people making inappropriate and incorrect decisions at a later point.

OORA

BERARD

Verification, Validation, And SQA Are Always Important

- Verification answers the question: "are we solving the problem correctly?" Validation answers the question: "are we solving the correct problem?"
- Software quality assurance ensures that we are addressing some aspect of the software life-cycle in an appropriate and approved manner.
- These processes will be done continually throughout the OORA process.

BERARD

OORA

BERARD

The OORA Process

- ❒ Identify the Sources of Requirements Information.
- ❒ Characterize the Sources of Requirements Information.
- ❒ Identify Candidate Objects.
- ❒ Build Object-Oriented Models of Both the Problem and Potential Solutions, As Necessary.
- ❒ Re-Localize the Information Around the Appropriate Candidate Objects.
- ❒ Select, Create, and Verify OCSs, Subsystem Specifications, and Systems of Objects Specifications.
- ❒ Assign OCSs, Subsystem Specifications, and Systems of Objects Specifications to the Appropriate Section of the OORS.
- ❒ Develop and Refine the Qualifications Section.
- ❒ Develop and Refine the Precise and Concise System Description.

OORA

BERARD

Identifying Sources of Requirements Information

Very seldom are requirements for a given project contained in a single, self-contained document. The OORA analyst must identify all valid, worthwhile sources of requirements information. These sources can include, for example:

- ✔ Pre-existing requirements documents
- ✔ Standards documentation
- ✔ Knowledgable people
- ✔ Previously existing software, including prototypes
- ✔ Descriptions of "real world" systems of objects

OORA

Characterizing Sources of Requirements Information

When characterizing sources of requirements information, we are interested in two things:

- ✔ The characteristics of the source itself
- ✔ The characteristics of the information which the source can provide

OORA

Characteristics of the Source

The following are important considerations when attempting to characterize the source of requirements information:

- ✔ The credibility of the source
- ✔ The ease of access the OORA analyst will have to the source
- ✔ The level of authority associated with the source
- ✔ The types of information which this source can provide
- ✔ The responsiveness of the source
- ✔ The longevity of the source

OORA

BERARD

Characteristics of the Information Provided by the Source

The following are important considerations when attempting to characterize the information provided by a source of requirements information:

- ✔ The form of the information provided, e.g., textual, graphical, verbal, executable, and machine-readable
- ✔ The completeness of the information provided by the source
- ✔ Identifying which specific aspects of the requirements are covered by the information
- ✔ Determining how current the provided information is
- ✔ Determining the volatility of the information

OORA

BERARD

Characteristics of the Information Provided by the Source (Continued)

- ✔ The relevance of the information
- ✔ How the information can be verified
- ✔ The availability of the information
- ✔ The understandability of the information
- ✔ The importance and priority of the information
- ✔ The interrelationships among the pieces of information, e.g., how will a change in one piece of information impact another piece of information

BERARD

OORA

BERARD

Identification of Candidate Objects, Classes, Systems of Objects, and Subsystems

Even before one has a fairly complete set of requirements information, one can begin to identify candidate objects. However, the OORA analyst should be reluctant to finalize the definition of any object-oriented item until there is a high degree of confidence that all the relevant information relating to that item has been identified.

OORA

BERARD

Short Maps

In the process of identifying candidate object-oriented items, the OORA analyst should keep a record of where a particular piece of information was found. The collection of these records for all the object-oriented items is referred to as a "short map." The form of a short map is very similar to an index, i.e., the name of each object-oriented item is listed (typically in alphabetical order), followed by pointers indicating where information on the item can be found.

OORA

BERARD

Building Object-Oriented Models of the Problem and Proposed Solutions

Some analysts find it useful to build object-oriented models of parts of the problem and/or solution prior to identifying all of the object-oriented items which comprise the system. The form of these models may be textual, graphical (e.g., Petri net graphs), or a mixture of both. These models may be used later in the construction of the applications-specific section of the object-oriented requirements specification.

OORA

BERARD

Relocalizing Information Around the Object-Oriented Items

This step requires that we gather everything we know about a particular object, class, system of objects, or subsystem, and place all this information in one place. In the case of a class, instance, or (some) system of objects, we refer to the relocalized information as a potential "OCS precursor." Exactly how we will treat subsystems requires further analysis.

OORA

Each Object Must Be Associated With a Class

- If we have identified any instances or systems of objects, each of these must be associated with a specific class. In fact, we will often use this information to better define the class.
- Although we typically only define OCSs for classes, if a language allows us to define instances without first defining a class, and there is no compelling reason to define a class for the instance we may occasionally create an OCS for an instance or system of objects.

OORA

Defining Subsystems

- A subsystem is a collection of program units which exports an object-oriented capability, e.g., windows, menus, switches, and panels. A subsystem may be built around a single class, or it may involve many different, but highly-related classes. There need not be a direct or indirect connection between any two components in the same subsystem.
- Subsystems are to classes as classes are to, say, operations.

OORA

The Selection, Creation, and Verification of OCSs and Subsystems

At this point in the OORA process, after examining the OCS precursors and tentative subsystem descriptions, the OORA analyst should have enough information to:

- ✔ Select a previously-defined instance or class (which has been documented with an OCS) to use for a particular instance or class which has been documented in the form of an OCS precursor
- ✔ Use the information available from a particular OCS precursor to define a new instance or class, and document that instance or class with an OCS

OORA

The Selection, Creation, and Verification of OCSs and Subsystems (Continued)

- ✔ Use the information available from a particular tentative subsystem description to select a previously-defined subsystem
- ✔ Use the information available from a particular tentative subsystem to create a new subsystem, or to extend a currently-existing subsystem

OORA

What About Objects and Other Application-Specific Information

At this same point, the OORA analyst will also have information regarding some of the objects which are instances of the defined classes, and other application-specific information. This information is not to be lost or discarded. Rather, it is to be placed in the appropriate sections of the application-specific section of the OORS.

OORA

All Information Must Be Verified and Quality Assured

Each of the classes, instances, systems of objects, and subsystems should be explicitly verified and quality assured. The same should be done for the application-specific information associated with these items.

OORA

BERARD

Placing Items In the Appropriate Section of the OORS

- Each OCS, subsystem specification, and system of objects specification must be assigned to the proper section (object-general or application-specific) of the OORS. The OORA analyst should take care to make sure that the items are properly organized.
- If the OORA analyst has not already done so, any application-specific information should be placed into the proper subsection of the application-specific section.

OORA

BERARD

Development of the Qualifications Section

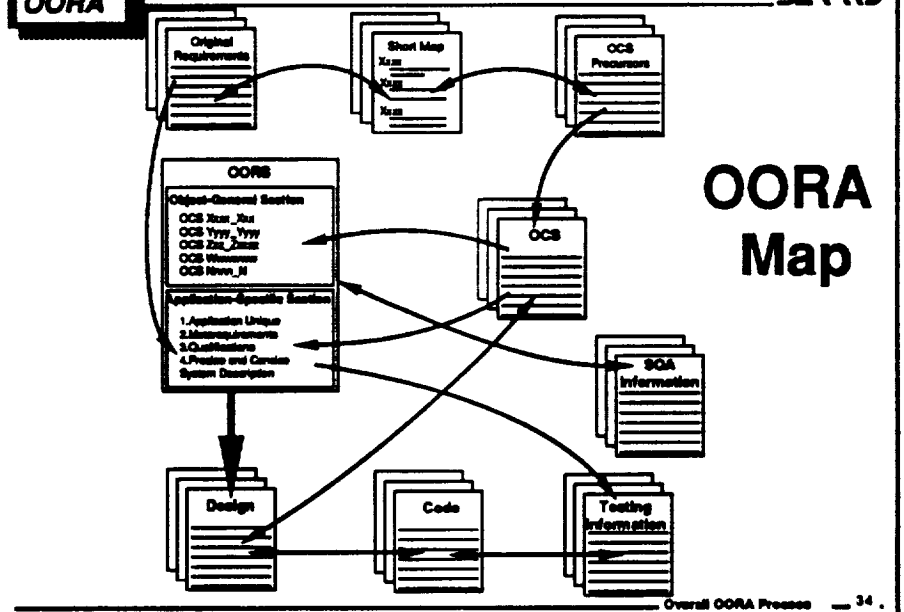
The creation, verification, and quality assurance of the qualifications subsection of the application-specific section of the OORS usually continues throughout the OORA process. Further, it is not until the precise and concise system description is finalized that we can finalize the qualifications section.

OORA

Development of the Precise and Concise System Description

Like the qualifications section, the precise and concise system description is often addressed throughout the OORA process. Once we are comfortable with our system description (i.e., it is complete, verified, and quality assured), we can verify and quality assure the entire OORS.

OORA



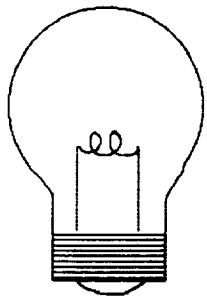
OORA Map

Object and Class Specification

Class: Lamp

1.0 Precise and Concise Description

1. A lamp is the abstraction of a simple lamp that can either be turned on (illuminated) or turned off (extinguished).



Lamp

2. Instances of this class require two operations, i.e., one which will allow a given lamp to communicate to the "outside world" that it has been "turned on," and one which will allow a given lamp to communicate to the "outside world" that it has been "turned off."

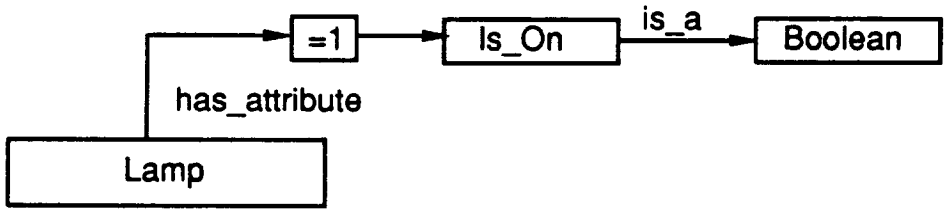
3. The suffered operations for a lamp are: turn the lamp on, turn the lamp off, Assign (the state of one lamp to another), and Is_On (is a given instance of this class in the "on" state). Since a lamp stays either on or off until instructed to change its state, the states of a lamp are persistent.

4. The lamp class will export no constants or exceptions.

2.0 Graphical Representations

2.1 Static Representations

2.1.1 Semantic Networks



2.1.2 Notes on the Semantic Networks

1. Two different instances of this class are equal if they are both in (or, are both not in) the "on" state.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

5.2 Exceptions

1. The exception Lamp_Not_Found will be raised if there is a request to turn a specific lamp on, and there is no lamp in the panel which corresponds to the specific lamp referred to in the request.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

7. The exception Lamp_Not_Found is associated with a floor arrival panel.

2.0 Graphical Representations

2.1 Static Representations

2.1.1 Semantic Networks

2.1.2 Notes on the Semantic Networks

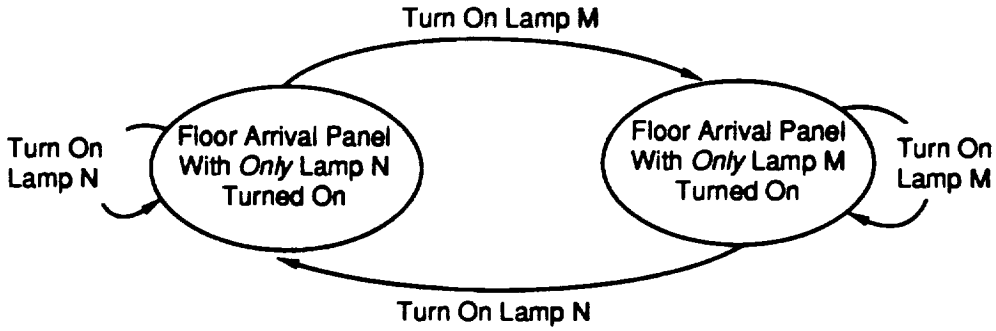
1. From the outside view, there is no discernable structure or attributes for a floor arrival panel.

2. Though it seems obvious that the floor arrival panel deals with lamps, these objects can neither be detected or affected directly from the "outside". It is for that reason that these objects do not appear on the semantic net.

2.2 Dynamic Representations

2.2.1 State Transition Diagrams

2.2.1.1 State Transition Diagrams for Non-Spontaneous State Changes



2.2.1.1.1 Notes on State Transition Diagrams for Non-Spontaneous State Changes

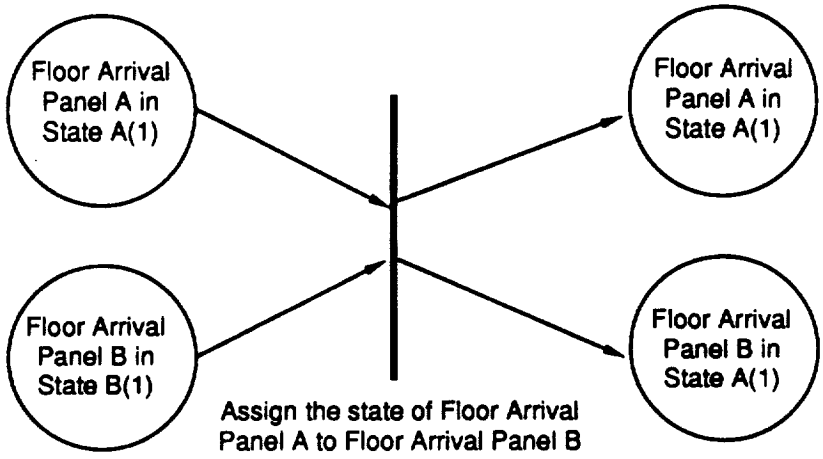
1. The states shown in the diagram cannot be interrogated. Specifically, there are no operations provided that will allow a client of the abstraction to determine if a particular floor arrival panel lamp is on or off.

2. A client may not interact *directly* with a specific floor arrival panel lamp. Clients deliver a request to turn a particular floor arrival panel lamp on or off via the Turn_On_Lamp operation. It is the computer circuitry in the floor arrival panel which will actually turn a lamp on or off, or leave a lamp in its particular state.

3. Since a given lamp will only be turned on or off based on a specific request, a floor arrival panel with any given lamp lit, and all others not lit, represents a persistent state.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

4. **Assign**: This constructor operation copies the state of one floor arrival panel object to another instance of the same class. Since the Assign operation produces an exact copy of an existing floor arrival panel, the resulting copy may be in any one of the states shown in the STD. The Petri Net Graph representation of the Assign operation is:



3.0 Operations

3.1 Required Operations

1. There are no required operations for this class.

3.2 Suffered Operations

Operation	Method
• Turn_On_Lamp	• Turns on the lamp corresponding to the given destination. If the lamp is already on, then there is no effect. This operation also turns off any other lamp which may be lit.
• Assign	• The state of one instance of this class to another instance of the same class.

4.0 State Information

1. The state for a floor arrival panel may be defined as which a single lamp (indicating a particular destination (floor)) is on, i.e., all other lamps must, by definition, be off. (Obviously, you cannot be at two different destinations simultaneously.) Note that there is no way for the client of the floor arrival panel to interrogate this state.

5.0 Constants and Exceptions

5.1 Constants

1. This class will not provide any constants.

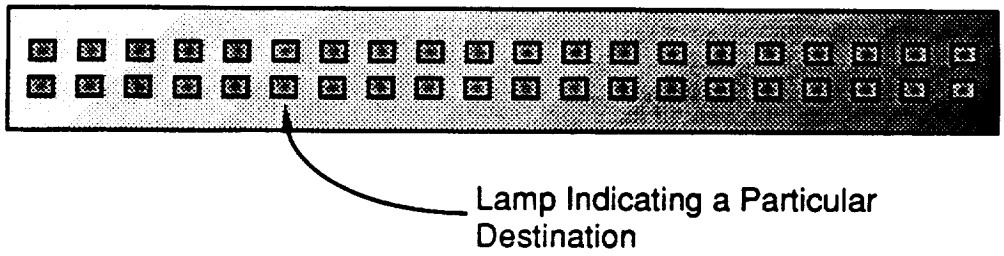
18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

Object and Class Specification

Class: Floor_Arrival_Panel

1.0 Precise and Concise Description

1. Conceptually, a floor arrival panel is a panel containing a number of lamps (typically one lamp for each destination (floor)). The floor arrival panel also contains some computer processing capability. This computer processing capability allows the floor arrival panel to turn a particular lamp on, based on a request, and to automatically ensure that all other lamps are off.



2. At any one time, only one lamp in the floor arrival panel may be lit (i.e., on). A given lamp in the floor arrival panel becomes lit (i.e., is turned on) based on an invocation of the Turn_On_Lamp operation. Once a lamp becomes lit, it stays lit until the floor arrival panel receives a request to turn another lamp on. Once a particular lamp is lit, any additional requests for the lamp to be lit are ignored. No facility is provided to determine the state of individual lamps contained in the floor arrival panel.

3. Obviously, any request to turn a given lamp on must contain some way of uniquely identifying the specific lamp. If the lamp identified in the request is not contained in the floor arrival panel, the exception Lamp_Not_Found will be raised.

4. The suffered operation for the floor arrival panel is "Turn_On_Lamp" (for a given destination).

5. Users of the Floor_Arrival_Panel class must also supply:

   - a class with discrete scalar values which will be used to uniquely identify destinations, i.e., Destination_ID, and

   - a value of this class which will represent the largest permissible value for a destination which can be specified by floor arrival panels which are instances of the Floor_Arrival_Panel class. Valid destinations will be represented by all values of the class Destination_ID from the smallest value up to, and including, the specified largest permissible value for a destination.

6. The state for a floor arrival panel may be defined as which particular lamp is on.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

Note that there is no way for the client of the destination panel to interrogate these states. Note further, that only one button in a given destination panel may be pressed at one time. If an attempt is made to press two, or more, destination buttons simultaneously, and at least of the destination buttons has a corresponding lamp which is not lit, the destination panel computer circuitry will select a single destination which will be transmitted via the "Signal" operation.

5.0 Constants and Exceptions

5.1 Constants

1. This class will not provide any constants.

5.2 Exceptions

1. The exception Lamp_Not_Found will be raised if a "lamp on/off request" is received and there is no lamp in the panel which corresponds to the specific lamp referred to in the request.

3. Pressing a destination button whose corresponding lamp is already lit will have no effect.

4. If an attempt is made to simultaneously press two, or more, destination buttons will result in the following:

   - All pressings of destination buttons whose corresponding lamps are already lit, will be ignored.

   - If two, or more, destination buttons, whose corresponding lamps are not lit, are pressed simultaneously, one of the destinations will be picked in a non-deterministic way, and notification of a request for that destination alone will be sent via the "Signal" operation.

3.0 Operations

3.1 Required Operations

Operation	Method
• Signal	• Alerts the destination panel's client that a button corresponding to a specific destination has been pressed.

3.2 Suffered Operations

Operation	Method
• Turn_On_Lamp	• Turns on the lamp corresponding to the given destination. If the lamp is already on, then there is no effect.
• Turn_Off_Lamp	• Turns off the lamp corresponding to the given destination. If the lamp is already off, then there is no effect.
• Assign	• The state of one instance of this class to another instance of this class.

4.0 State Information

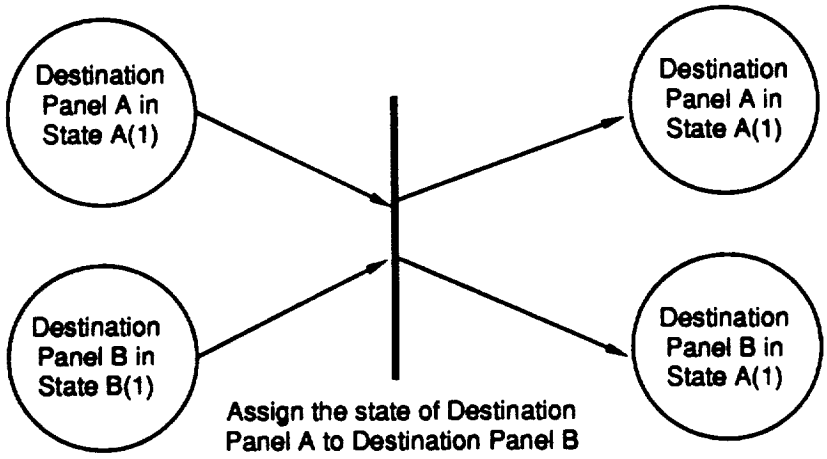
1. The state for a destination panel may be defined as:

   - the sum of the states of the lamps (i.e., on or off) which are associated with destinations (these states are persistent), and

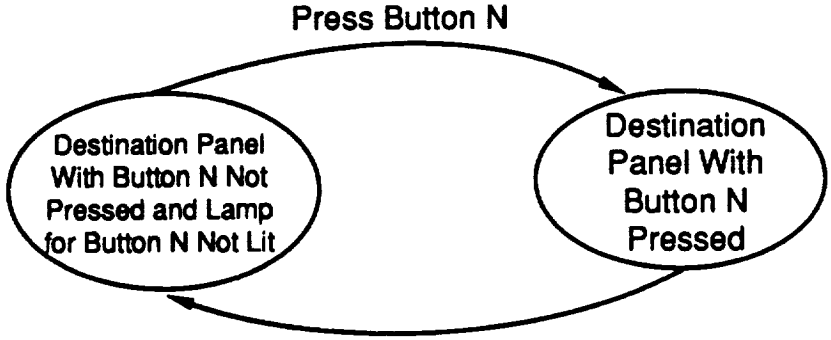
   - whether a destination button, whose corresponding lamp is not lit, has been pressed (this state is not persistent).

3. Since a given lamp will only be turned on or off based on a specific request, a destination panel with any given number of lamps lit and not lit represents a persistent state.

4. **Assign**: This constructor operation copies the state of one destination panel object to another instance of the same class. Since the Assign operation produces an exact copy of an existing destination panel, the resulting copy may be in any one of the states shown in the STD. The Petri Net Graph representation of the Assign operation is:



2.2.1.2 State Transition Diagrams for Spontaneous State Changes



2.2.1.2.1 Notes on State Transition Diagrams for Spontaneous State Changes

1. Clients of the destination panel cannot "press" any of the destination panel buttons.

2. After a button is pressed, the destination panel immediately returns to the state where that button is not pressed, i.e., a destination panel with any given button pressed is a highly non-persistent state.

2. Clients of the destination panel are notified that a particular destination panel button has been pressed via the "Signal" operation. It is the computer circuitry in the destination panel which will actually provide the notification that a particular destination button has been pressed.

2.0 Graphical Representations

2.1 Static Representations

2.1.1 Semantic Networks

2.1.2 Notes on the Semantic Network

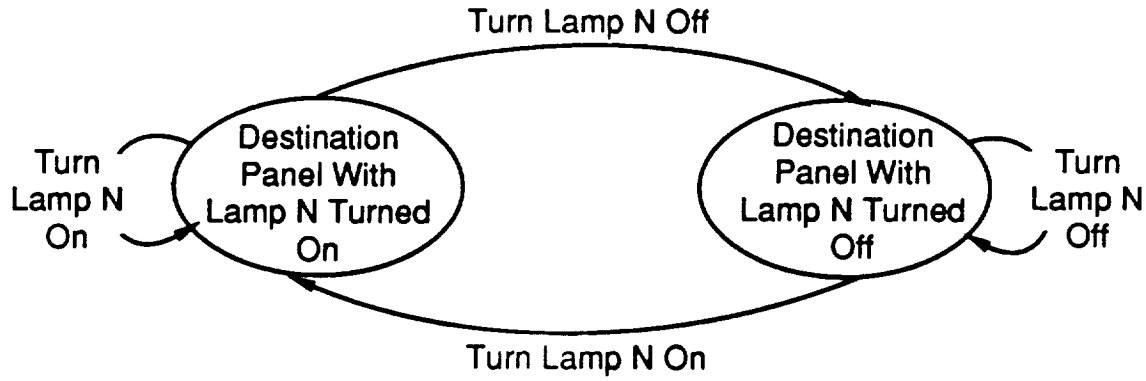
1. From the outside view, there is no discernable structure or attributes for a destination panel.

2. Though it seems obvious that the destination panel deals with buttons and lamps, these objects can neither be detected or affected directly from the "outside". It is for that reason that these objects do not appear on the semantic net.

2.2 Dynamic Representations

2.2.1 State Transition Diagrams

2.2.1.1 State Transition Diagrams for Non-Spontaneous State Changes



2.2.1.1.1 Notes on State Transition Diagrams for Non-Spontaneous State Changes

1. The states shown in the diagram cannot be interrogated. Specifically, there are no operations provided that will allow a client of the abstraction to determine if a particular destination panel lamp is on or off.

2. A client may not interact *directly* with a specific destination panel lamp. Clients deliver a request to turn a particular destination panel lamp on or off to the "lamp on/off request port." It is the computer circuitry in the destination panel which will actually turn a lamp on or off, or leave a lamp in its particular state.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

4. The destination panel must notify the outside world that a particular destination button has been pressed. It does this through a required operation, Signal. Each destination button is associated with a specific lamp, and the panel has the (internal) capability of determining which of its lamps are lit. If someone presses a destination button for which the associated lamp that is already lit, no (new) notification is passed to the outside world. No facility is provided to determine the state of individual buttons contained in the destination panel.

5. Obviously, any notification (to the outside world) that a specific destination button has been pressed (i.e., done through the "Signal" operation) must contain some way of uniquely identifying the specific desired destination.

6. Since, to all who must deal with the destination panel abstraction, the destination panel appears to be changing its state spontaneously (i.e., specific destinations will be periodically requested), the destination panel is an "object with life."

7. The required operation for the destination panel is: "Signal" (the user of a destination panel that a button has been pressed).

8. The suffered operations for the destination panel are "Turn_On_Lamp" (for a given destination), "Turn_Off_Lamp" (for a given destination), and Assign (the state of one destination panel to another destination panel).

9. Users of the Destination_Panel class must also supply:

   - a class with discrete scalar values which will be used to uniquely identify destinations, i.e., Destination_ID, and

   - a value of this class which will represent the largest permissible value for a destination which can be specified by destination panels which are instances of the Destination_Panel class. Valid destinations will be represented by all values of the class Destination_ID from the smallest value up to, and including, the specified largest permissible value for a destination.

10. The state for a destination panel may be defined as:

    - the sum of the states of the lamps (i.e., on or off) which are associated with destinations (these states are persistent), and

    - whether a destination button, whose corresponding lamp is not lit, has been pressed (these states are not persistent).

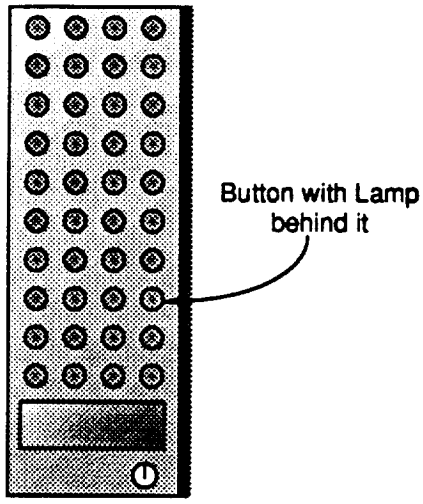
11. The exception Lamp_Not_Found is associated with a destination panel.

Object and Class Specification

Class: Destination_Panel

1.0 Precise and Concise Description

1. Conceptually, a destination panel is a panel containing a number of destination buttons (typically one for each reachable destination), a number of lamps (typically one lamp for each destination button) and, potentially, other devices. The destination panel also contains some computer processing capability. This computer processing capability allows the destination panel to turn particular lamps on and off based on requests, and to inform the outside world when a particular destination button has been pressed.



2. At any one time, any number of lamps in the destination panel may be lit (i.e., on). A given lamp in the destination panel becomes lit (i.e., is turned on) based on an invocation of the Turn_On_Lamp operation. Once a lamp becomes lit, it stays lit until the destination panel receives a request to turn that lamp off, i.e., via an invocation of the Turn_Off_Lamp operation. Likewise, a lamp remains off until a request is received to turn it on. Once a particular lamp is lit (or turned off), any additional requests for the lamp to be lit (or turned off) are ignored. No facility is provided to determine the state of individual lamps contained in the destination panel.

3. Obviously, any request to turn a given lamp on or off must contain some way of uniquely identifying the specific lamp, and whether that lamp is to be turned on or off. If the lamp identified in the request is not contained in the destination panel, the exception Lamp_Not_Found will be raised.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

• Append	• a given bounded list to the specified bounded list
• Break_Up	• a given bounded list, at a specified position, into two specified sublists

4.0 State Information

1. The state information for a bounded list is:
   - The current number of elements contained in the bounded list
   - Whether a given bounded list is empty
   - Whether a given bounded list is full
   - The specific elements stored in a given bounded list
   - The specific order of the elements stored in a given bounded list
2. Note that while most state information for a given bounded list may be determined via the invocation of a single operation, the specific order of the elements in a given bounded list may require the invocation of many operations to be determined accurately.

5.0 Constants and Exceptions

5.1 Constants

1. This class will provide the constant "empty list".

5.2 Exceptions

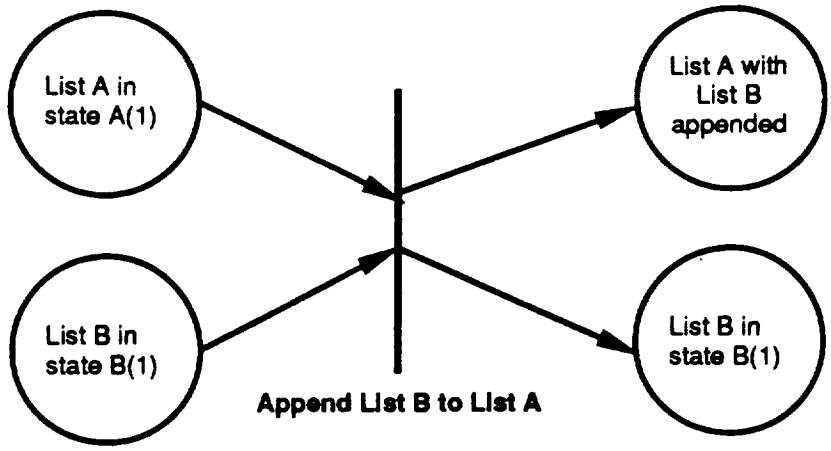
1. This class will provide the following exceptions:
   - The exception **Overflow** will be raised if a user tries to: insert an element into a *full* list, copy a list into another list which has a *smaller upper limit* of the number of elements than the former does, append a list into another list whose number of unused spaces is *less* than the current number of elements in the former list, or break up a list into lists whose total maximum lengths are smaller than the current length of the list to be broken up.
   - The exception **Underflow** will be raised if a user tries to remove an element from an *empty* list.
   - The exception **Element_Not_Found** will be raised if a user specifies a *non-existent element* or a *non-existent element location.*

• Test for equality	• of one of the elements to be placed in the list with another element of the same class
• Assignment	• of the value of an instance of the class used to "count" the number of elements in a given bounded list to another instance of the same class
• Set to "zero"	• for the class used to indicate the length of the bounded list
• Increment "by one"	• the value of an object of the class used to indicate the length of the bounded list
• Decrement "by one"	• the value of an object of the class used to indicate the length of the bounded list

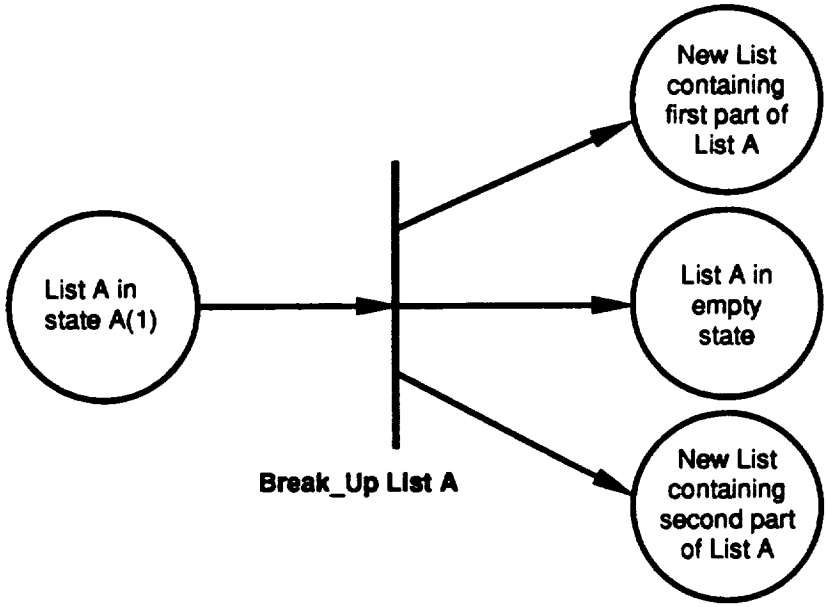
3.2 Suffered Operations

Operation	Method
• Clear	• the contents of a given bounded list, i.e., removes all elements from the specified bounded list
• Insert	• a given element into a given bounded list
• Remove	• a given element from a specified bounded list
• Length_Of	• a specified bounded list
• Copy	• the contents of a given bounded list into another specified bounded list producing a bounded list identical in contents to the original bounded list
• "="	• tests for the equality of two specified bounded lists. Two bounded lists are equal if the current lengths of both lists are the same, and if the values of the corresponding elements in both lists are the same.
• Is_Empty	• determines if a given bounded list is empty
• Is_Full	• determines if a given bounded list is full, i.e., it contains the maximum number of elements

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652



d. **Break_Up**: This constructor operation splits a given list at a specified location and puts the results in two sublists. The original bounded list will become an empty list. Depending on both the state of the original bounded list and the location specified for breaking the list, each of the two sublists may be in any of the states shown in the STD.



3.0 Operations

3.1 Required Operations

Operation	Method
• Assignment	• of the value of an element to be placed in the list to another element of the same class

2.2.1.1.1 Notes on State Transition Diagrams for Non-Spontaneous State Changes

1. **Node A** represents an empty bounded list. **Node B** represents a bounded list with one element. **Node C** represents a full bounded list. **Node E** represent a bounded list containing some elements.

2. The state transition "insert maximum number of elements " may be accomplished by repeated application of the "insert" operation.

3. The state transition "remove all but one element " may be accomplished by repeated application of the "remove" operation.

4. The selector operation **Length_Of** may be used to determine if a bounded list is empty, contains one element, or contains some elements.

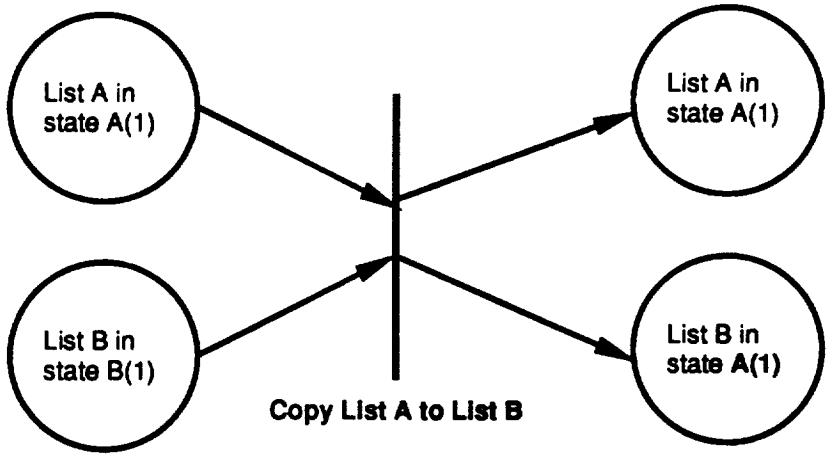
5. The selector operation **Is_Full** may be used to determine if a bounded list contains its maximum allotted number of elements.

6. The selector operation **Is_Empty** may be used to determine if a bounded list contains no elements.

7. The following operations all require two or more bounded lists, and thus cannot be accurately shown on a single STD:

   a. "=" : This selector operation is the test for equality of two bounded lists. This operation compares the states of two different bounded lists.

   b. **Copy**: This constructor operation copies one entire bounded list to another bounded list. Since the Copy operation produces an exact copy of an existing bounded list, the resulting copy may be in any one of the states shown in the STD.



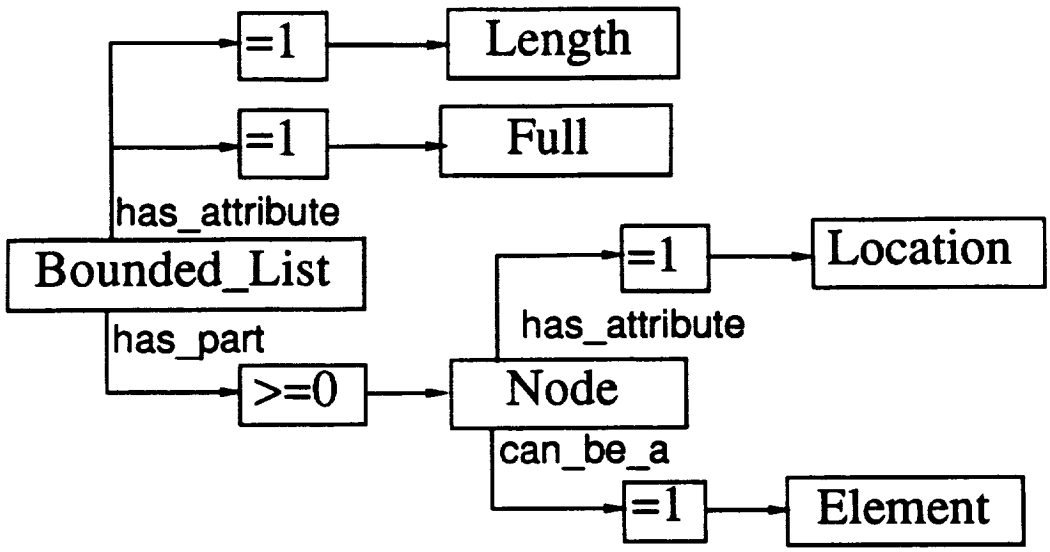
   c. **Append**: This constructor operation appends one bounded list to another bounded list. The Append operation will result in a bounded list which may be in any one of the states shown in the STD.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

2.0 Graphical Representations

2.1 Static Representations

2.1.1 Semantic Networks



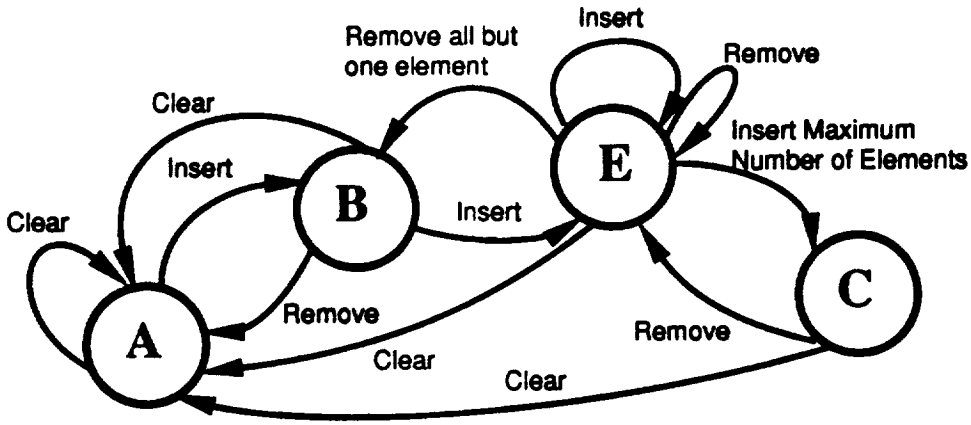
2.1.2 Notes On the Semantic Networks

1. The attribute "(is the bounded list) empty" can be determined directly from the "length (of the bounded list)" attribute. Hence, we do not show this attribute separately.

2.2 Dynamic Representations

2.2.1 State Transition Diagrams

2.2.1.1 State Transition Diagrams for Non-Spontaneous State Changes



Object and Class Specification

Class: Bounded List

1.0 Precise and Concise Description

1. A *linear list* (or simply, list) is defined to be "a set of $n >= 0$ nodes X[1], ... X[n] whose structural properties essentially involve only the linear (one-dimensional) relative positions of the nodes: ..., if $n > 0$, X[1] is the first node; when $1 < k < n$, the k th node X[k] is preceded by X[k - 1] and followed by X[k + 1]; and X[n] is the last node." (See [Knuth, 1973].) The number of elements (n) is called the *length* of the list. If $n = 0$, then the list is said to be *empty*.

2. A **bounded** list is a list which has a *fixed limit* on the maximum number elements that can be stored in it. A user will have to specify the maximum length of a bounded list when a list object is declared.

3. The following is a list of operations that can be applied to a bounded list: clear a list, insert an element into a list, remove an element from a list, find out the current length of a list, copy a list to another list, check whether one list is equal to another list, check whether a list is empty, check whether a list is full, append one list to the end of another list, and break a list into two parts.

4. The user is not concerned with the type of elements that can be put in the list. The class of the elements to be placed in the list must be supplied by users of this class. The following required operations for the list must be applicable to the class of the elements to be placed in the list. The required operations are: assignment (of the value of one element to another), and test for equality (of the value of one element with another).

5. Users of the bounded list class must also supply a class which will be used to "count" the number of elements in a given bounded list, and which will also be used to identify locations within the list. The required operations for this class are assignment (of one value of an instance of this class to another), set to "zero" (set the value of an instance of this class to indicate no elements in a given bounded list), increment ("by one"), and decrement ("by one").

6. This class will export the exceptions Underflow, Overflow, and Element_Not_Found.

7. This class will export the constant "empty list."

[Knuth, 1973]. D.E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, Massachusetts, 1973.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

• Value_Of	• Returns the state of the Vendor Number component of a bid.
• Value_Of	• Returns the state of the Bid Date component of a bid.
• Value_Of	• Returns the state of the FOB component of a bid.
• Value_Of	• Returns the state of the Freight Cost component of a bid.
• Value_Of	• Returns the state of the Estimated Lead Time component of a bid.
• Value_Of	• Returns the state of the Special Instructions component of a bid.
• Value_Of	• Returns the state of the Vendor Terms component of a bid.
• Value_Of	• Returns the state of the Total Weight component of a bid.
• Assign	• Assigns the state of one Bid object to another.

4.0 State Information

The state of an bid is the sum of the states of all its component parts. Each component part's state is independent of the state of any other component part.

5.0 Constants and Exceptions

5.1 Constants

1. This class will not provide any constants.

5.2 Exceptions

1. This class will not provide any exceptions.

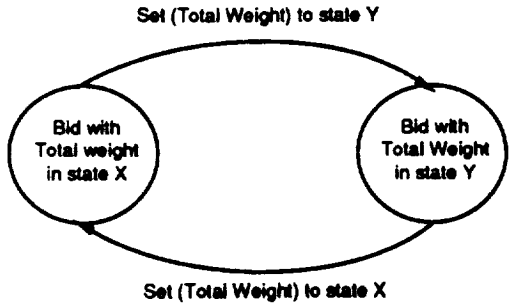
18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

• Assign	• Assigns the state of one Vendor Terms object to another.
• Assign	• Assigns the state of one Total Weight object to another

3.2 Suffered Operations

Operation	Method
• Set	• Sets the state of the Bid Item List component of a bid.
• Set	• Sets the state of the Buyer Location component of a bid.
• Set	• Sets the state of the RFQ Number component of a bid.
• Set	• Sets the state of the Vendor Number component of a bid.
• Set	• Sets the state of the Bid Date component of a bid.
• Set	• Sets the state of the FOB component of a bid.
• Set	• Sets the state of the Freight Cost component of a bid.
• Set	• Sets the state of the Estimated Lead Time component of a bid.
• Set	• Sets the state of the Special Instructions component of a bid.
• Set	• Sets the state of the Vendor Terms component of a bid.
• Set	• Sets the state of the Total Weight component of a bid.
• Value_Of	• Returns the state of the Bid Item List component of a bid.
• Value_Of	• Returns the state of the Buyer Location component of a bid.
• Value_Of	• Returns the state of the RFQ Number component of a bid.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652



2.2.1.1.1 Notes on the State Transition Diagrams for Non-Spontaneous State Changes

1. There is a selector and a constructor operation provided for each component part of a bid. Each constructor operation changes the state of only one component part.

3.0 Operations

3.1 Required Operations

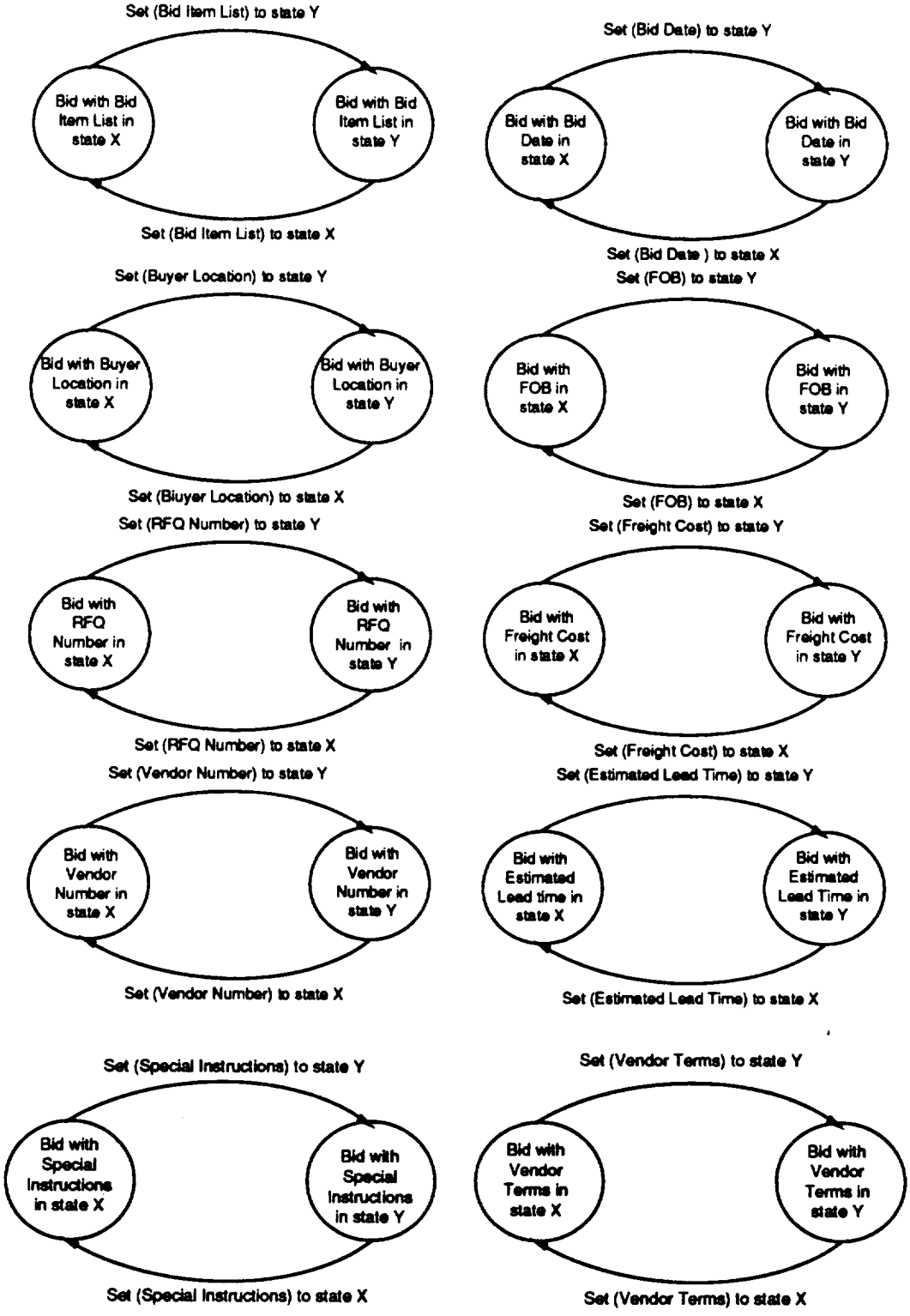
Operation	Method
• Assign	• Assigns the state of one Bid Item List object to another.
• Assign	• Assigns the state of one Buyer Location object to another.
• Assign	• Assigns the state of one RFQ Number object to another.
• Assign	• Assigns the state of one Vendor Number object to another.
• Assign	• Assigns the state of one Bid Date object to another.
• Assign	• Assigns the state of one FOB object to another.
• Assign	• Assigns the state of one Freight Cost object to another.
• Assign	• Assigns the state of one Estimated Lead Time object to another.
• Assign	• Assigns the state of one Special Instructions object to another.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

2.2 Dynamic Representations

2.2.1 State Transition Diagrams

2.2..1.1 State Transition Diagrams for Non-Spontaneous State Changes

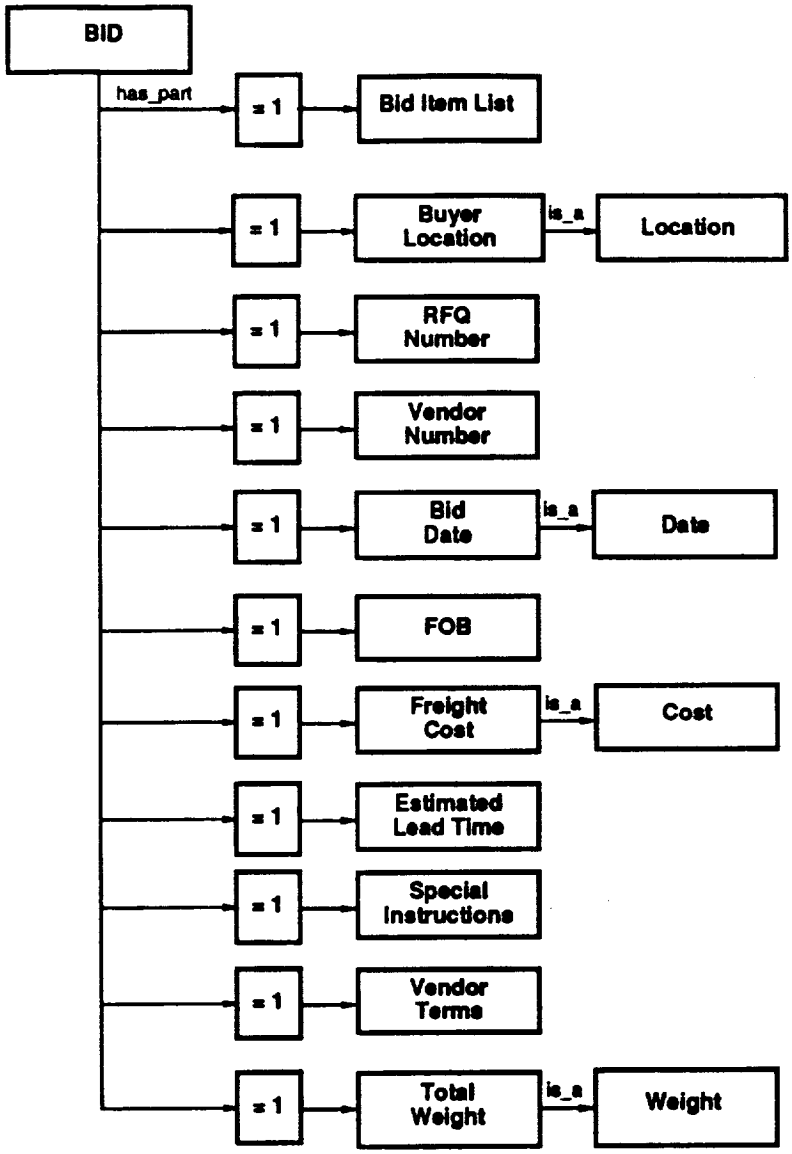


18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

2.0 Graphical Representations

2.1 Static Representations

2.1.1 Semantic Networks



2.1.2 Notes on the Semantic Networks

1. The component parts of the bid are independent, i.e. a change in the state of any component will not change the state of any other component part.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

Object and Class Specification

Class: Bid

1.0 Precise and Concise Description

1. A bid represents a document containing all information that a vendor would supply to a customer in response to a request for quote (RFQ). The information supplied by the document includes the following: a list of the items being supplied by the vendor, the location of the buyer, the RFQ number, the vendor number, the date of the bid, FOB, the freight costs for the bid, the estimated lead time required, any special instructions, the vendor's payment terms, and the total weight of the order.

2. The state of a bid is the state of its component parts.

3. The required operations for a bid are: Assign (one Bid Item List to another), Assign (one Buyer Location to another), Assign (one RFQ Number to another), Assign (one Vendor Number to another), Assign (one Bid Date to another), Assign (one FOB to another), Assign (one Freight Cost to another), Assign (one Estimated Lead Time to another),Assign (one Special Instructions to another), Assign (one Vendor Terms to another), and Assign (one Total Weight to another),.

4. The suffered operations for a bid are:

   a. Constructor operations are: Set (Bid Item List), Set (Buyer Location), Set (RFQ Number), Set (Vendor Number), Set (Bid Date), Set (FOB), Set (Freight Cost), Set (Estimated Lead Time), Set (Special Instructions), Set (Vendor Terms), and Set (Total Weight).

   b. Selector operations are: Value_Of (Bid Item List), Value_Of (Buyer Location), Value_Of (RFQ Number), Value_Of (Vendor Number), Value_Of (Bid Date), Value_Of (FOB), Value_Of (Freight Cost), Value_Of (Estimated Lead Time), Value_Of (Special Instructions), Value_Of (Vendor Terms), and Value_Of (Total Weight).

   c. Additional operation (for completeness) is Assign(one Bid to another).

5. There are no exceptions associated with a bid.

6. There are no constants associated with a bid.

7. The Bid class requires that eleven classes be imported to correspond to the following: Bid Item List, Buyer Location, RFQ Number, Vendor Number, Bid Date, FOB, Freight Cost, Estimated Lead Time, Special Instructions, Vendor Terms, and Total Weight. There are no restrictions placed on these imported classes.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

2. A Write_Only_Port is momentarily in a different state when it is writing values represented as bit patterns.

5.0 Constants and Exceptions

5.1 Constants

1. This class provides no constants

5.2 Exceptions

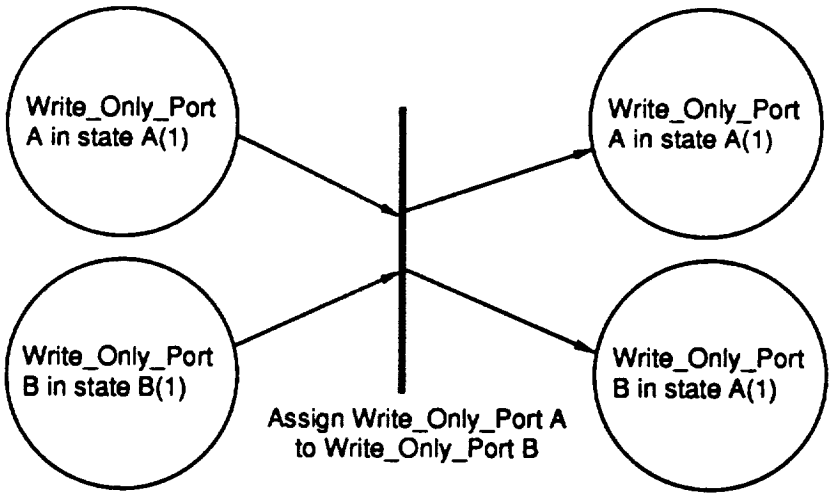
1. This class provides an exception Address_Not_Defined which is raised if an attempt is made to read from a port which has not been assigned to an address, or if a port which has not been assigned an address is queried as to its address.

2.2.1.1.1 Notes on State Transition Diagrams for Non-Spontaneous State Changes

1. A given Write_Only_Port object cannot be read from, or queried about its address until it has been assigned an address.

2. After writing a value the Write_Only_Port immediately returns to an inactive state.

3. The Assign operation requires two instances of the the class Write_Only_Port, and therefore cannot be shown on a state transition diagram. The Petri Net Graph for the Assign operation is:



3.0 Operations

3.1 Required Operations

1. The Write_Only_Port class has no required operations.

3.2 Suffered Operations

Operation	Method
• Set_Address	• Dynamically assigns an address to a Write_Only_Port
• Address_Of	• Returns the address of a port
• Assign	• Assigns the state of one Write_Only_Port object to another
• Write	• A value of the given integer class to a given Write_Only_Port

4.0 State Information

1. The address of a Write_Only_Port object can be changed and queried.

6. The *suffered* operations for the Write_Only_Port are Set_Address (dynamically set the address for the port), Address_Of (the specified port), Assign (one the value of one Write_Only_Port to another), and Write (a specified bit pattern to a given Write_Only_Port).

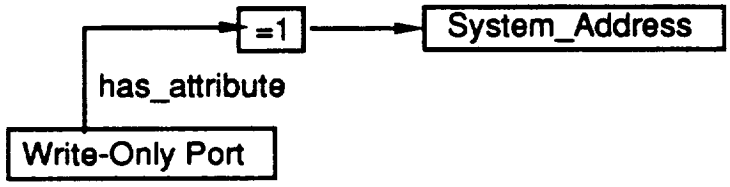
7. The Write_Only_Port class will provide no constants.

8. The Write_Only_Port class will provide an exception: Address_Not_Defined.

2.0 Graphical Representations

2.1 Static Representations

2.1.1 Semantic Networks

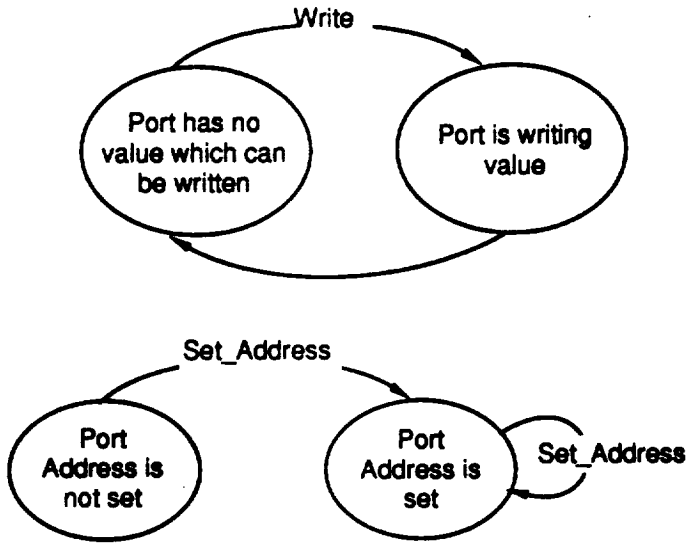


2.1.2 Notes on the Semantic Networks

2.2 Dynamic Representations

2.2.1 State Transition Diagrams

2.2.1.1 State Transition Diagrams for Non-Spontaneous State Changes



Object and Class Specification

Class: Write_Only_Port

1.0 Precise and Concise Description

1. A "port" is an abstraction of a highly-localized interface between two pieces of hardware in a (potentially embedded) computer system. A port is a place where information can be transferred into, out of, or to and from, a hardware component.

2. A port has several distinguishing characteristics:

   - an **address**. Every port in a system must be directly, or indirectly, addressable within the "address space" (i.e., the set of all allowable addresses) of the cpu (central processing unit) charged with dealing with the port.

   - a **width** (measured in bits). The width of a port refers to how many bits may be *simultaneously* read from, or written to, the port. It is assumed that the bits are contiguous.

   - whether it is **read-only**, **write-only**, or **read-write** (i.e., bi-directional). Often, ports are uni-directional, that is, they can either be read from or written to, but not both.

3. The purpose of the port abstraction is to provide a uniform interface for instances of other classes which use ports. Internally, ports deal with the unique characteristics of the hardware for which they were created. Externally, they present a constant and uniform interface for objects which must deal with ports.

4. Ports view information only as "bit patterns." For example, if a four-bit wide port provides the value 1011_2 , it places no special significance, or meaning, on this value. Interpretations of bit patterns are left to the clients of the port.

5. In reality, the Write_Only_Port class is a metaclass. Users of the Write_Only_Port class must supply:

   - a width (i.e., a non-zero, positive integer value) which will be used to set a fixed width (in bits) for all instances of the Write_Only_Port class,

   - an integer class which will be used to contain the values written by the port, and

   - a system address class, instances of which will be used to dynamically assign a given Write_Only_Port to a specific system address.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

2. The creation of aliases for instances of this class does change their states, i.e., they now have aliases. Therefore, part of the state information for an instance of this class is whether the instance has aliases.

5.0 Constants and Exceptions

5.1 Constants

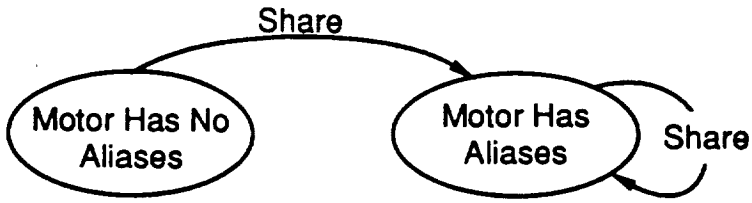
1. This class will provide no constants.

5.2 Exceptions

1. This class will provide the following exceptions:

   - Mechanical_Failure: raised if the motor is not able to respond to a request.

   - Not_Stopped: raised if the motor's rotor is rotating in a particular direction (e.g., clockwise), and an attempt is made to cause it to rotate in the opposite direction without first stopping the motor.

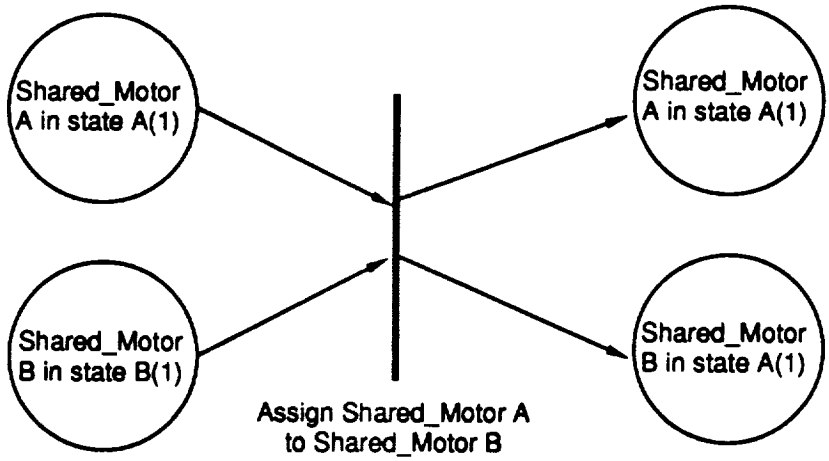


2.2.1.1.1 Notes on State Transition Diagrams for Non-Spontaneous State Changes

1. The operations: Is_Rotating_Clockwise, Is_Rotating_Counterclockwise, and Is_Stopped, are selector operations which can be used to determine if a given motor is in one of the states shown.

2. The operation Is_Shared can be used to determine if a given motor object has aliases.

3. The operation "assign" requires two instances of this class, and, thus, cannot easily be shown on a simple state transition diagram. The Petri Net Graph representation of this operation is:



3.0 Operations

3.1 Required Operations

Operation	Method
• Rotate_Clockwise	• Connects the motor with the necessary operations to rotate clockwise.
• Rotate_Counterclockwise	• Connects the motor with the necessary operations to rotate counterclockwise.
• Stop	• Connects the motor with the necessary operations to stop.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

• Is_Rotating_Clockwise	• Connects the motor with the necessary operations to determine if the motor's rotor is rotating clockwise
• Is_Rotating_Counterclockwise	• Connects the motor with the necessary operations to determine if the motor's rotor is rotating counterclockwise
• Is_Stopped	• Connects the motor with the necessary operations to determine if the motor's rotor is stopped

3.2 Suffered Operations

Operation	Method
• Rotate_Clockwise	• Causes the motor's rotor to rotate clockwise
• Rotate_Counterclockwise	• Causes the motor's rotor to rotate counterclockwise
• Stop	• Causes the motor's rotor to stop rotating
• Is_Rotating_Clockwise	• Returns true if the motor's rotor is rotating clockwise
• Is_Rotating_Counterclockwise	• Returns true if the motor's rotor is rotating counterclockwise
• Is_Stopped	• Returns true if the motor's rotor is not rotating
• Assign	• Assigns the state of one instance of this class to another instance of the same class
• Share	• Allows for share semantics, i.e., allows for the creation of aliases for instances of the motor class.
• Is_Shared	• Returns true if the given instance of the motor class has an alias.

4.0 State Information

1. The state information for a motor is:

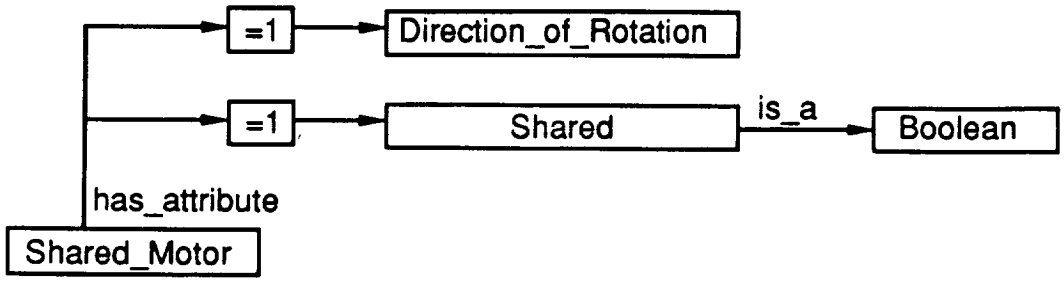
    - The direction of rotation, which typically assumes values of: clockwise, counterclockwise, and stopped.

8. There are no constants associated with the instances of the Shared_Motor chass.

2.0 Graphical Representations

2.1 Static Representations

2.1.1 Semantic Networks



2.1.2 Notes on the Semantic Networks

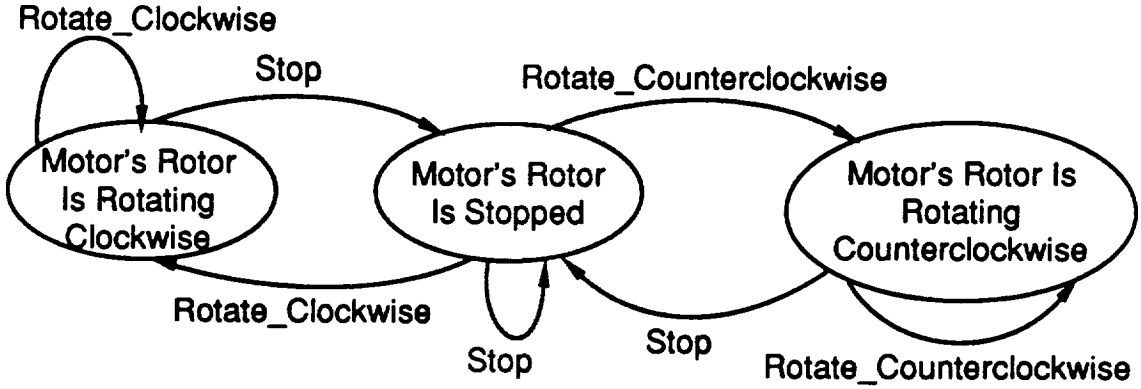
1. Direction of rotation can only assume one of three values: rotating clockwise, rotating counterclockwise, and stopped.

2. While we can determine whether or not a given instance of this class has aliases, we cannot determine *how many* aliases it has.

2.2 Dynamic Representations

2.2.1 State Transition Diagrams

2.2.1.1 State Transition Diagrams for Non-Spontaneous State Changes

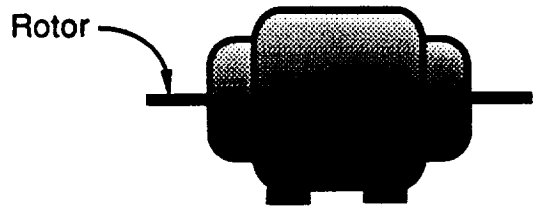


Object and Class Specification

Class: Shared_Motor

1.0 Precise and Concise Description

1. A motor is an abstraction of a physical motor device that converts energy into movement. Movement is delivered in the form of the rotation of the motor's rotor. A rotor may be viewed as a shaft which is part of a motor, and to which varying devices may be attached.



2. "Share semantics" allow for the creation of aliases for an object. This has the advantage of possible increases in time and space efficiency. However, it increases the possibility of actions with unintended results, e.g., unintentional deletion or alteration of an object through operations performed using the alias. (Instances of this class may have more than one alias.)

3. The suffered operations for a motor include movement operations (rotate_clockwise, rotate_counterclockwise, stop), operations for detecting movement (is_rotating_clockwise, is_rotating_counterclockwise, is_stopped), assign (the state of one instance of this class to another instance of the same class) and operations for share semantics (share and is_shared).

4. If a motor object is in the "stop" state then either the rotate_clockwise or rotate_counterclockwise operations may be accomplished. If the motor's rotor is rotating clockwise or counterclockwise, and rotation in the opposite direction is desired, the motor must first be stopped. If a motor is in a particular state, and an operation is invoked which would result in the motor maintaining that state, no state changes will occur, i.e., the operation will be ignored.

5. This motor abstraction represents a motor which produces movement of constant speed, i.e., it is incapable of varying speeds of rotation.

6. The motor contains state information about the current direction of rotation, or, more precisely, about the direction of rotation of the motor's rotor. The allowed states for direction of rotation are: clockwise, counterclockwise, and stopped. An additional piece of state information is whether a given motor object has aliases, i.e., is shared.

7. The exceptions for a shared motor object are Mechanical_Failure and Not_Stopped.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

3.0 Operations

3.1 Required Operations

Operation	Method
• Signal	• The action(s) clients of the Read_Only_Port want the port to take when information arrives at the port. This usually involves the actual transference of the information.

3.2 Suffered Operations

Operation	Method
• Set_Address	• Dynamically assigns an address to a Read_Only_Port
• Address_Of	• Returns the address of a port
• Assign	• Assigns the state of one Read_Only_Port object to another

4.0 State Information

1. The address of a Read_Only_Port object can be changed and queried.

2. A Read_Only_Port is an object with life. Specifically, it periodically and "spontaneously" produces values represented as bit patterns.

5.0 Constants and Exceptions

5.1 Constants

1. This class provides no constants

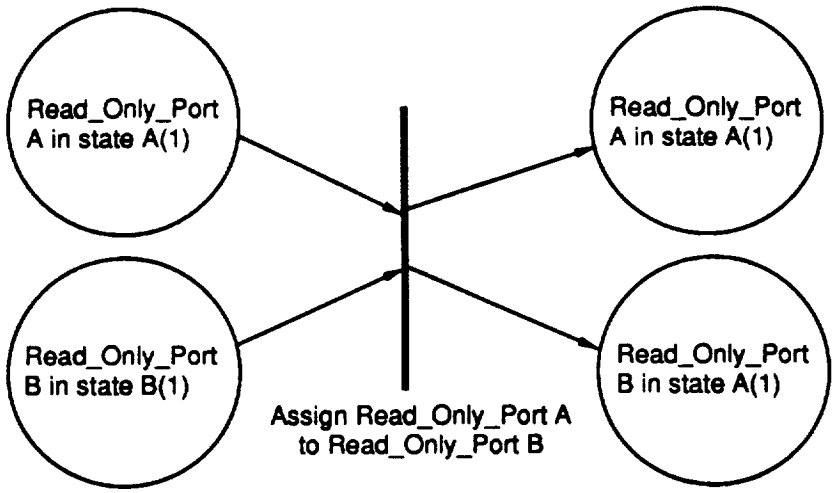
5.2 Exceptions

1. This class provides an exception Address_Not_Defined which is raised if an attempt is made to read from a port which has not been assigned to an address, or if a port which has not been assigned an address is queried as to its address.

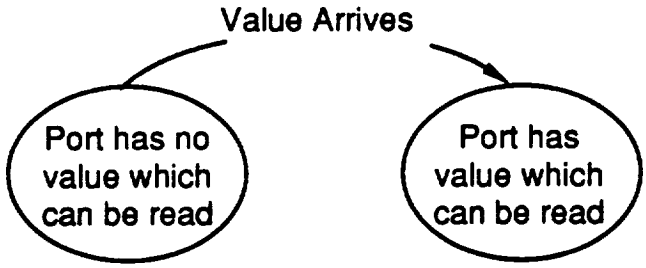
2.2.1.1.1 Notes on State Transition Diagrams for Non-Spontaneous State Changes

1. A given Read_Only_Port object cannot be read from, or queried about its address until it has been assigned an address.

2. The Assign operation requires two instances of the the class Read_Only_Port, and therefore cannot be shown on a state transition diagram. The Petri Net Graph for the Assign operation is:



2.2.1.2 State Transition Diagrams for Spontaneous State Changes



2.2.1.2.1 Notes on State Transition Diagrams for Spontaneous State Changes

1. When information arrives at the hardware port that information will be transferred to software clients of the port via a Signal operation. Thus, clients of a port will not have to poll the port.

Object and Class Specification

Class: Read_Only_Port

1.0 Precise and Concise Description

1. A "port" is an abstraction of a highly-localized interface between two pieces of hardware in a (potentially embedded) computer system. A port is a place where information can be transferred into, out of, or to and from, a hardware component.

2. A port has several distinguishing characteristics:

   - an **address**. Every port in a system must be directly, or indirectly, addressable within the "address space" (i.e., the set of all allowable addresses) of the cpu (central processing unit) charged with dealing with the port.

   - a **width** (measured in bits). The width of a port refers to how many bits may be *simultaneously* read from, or written to, the port. It is assumed that the bits are contiguous.

   - whether it is **read-only**, **write-only**, or **read-write** (i.e., bi-directional). Often, ports are uni-directional, that is, they can either be read from or written to, but not both.

3. The purpose of the port abstraction is to provide a uniform interface for instances of other classes which use ports. Internally, ports deal with the unique characteristics of the hardware for which they were created. Externally, they present a constant and uniform interface for objects which must deal with ports.

4. A Read_Only_Port is an "object with life," i.e., there is no software mechanism for changing some aspects of the state of the port. Clients of the Read_Only_Port, for example, cannot *force* the Read_Only_Port to provide them with information at any given time. They must wait for the port to provide them with information. Ports do not buffer information, i.e., if the information is not read when it becomes available, the information is lost.

5. Ports view information only as "bit patterns." For example, if a four-bit wide port provides the value 1011_2, it places no special significance, or meaning, on this value. Interpretations of bit patterns are left to the clients of the port.

6. The Read_Only_Port has one *required* operation ("Signal") which contains the actions the client wishes to accomplish when information arrives at the hardware port — usually includes the transference of that information.

7. The *suffered* operations for the Read_Only_Port are Set_Address (dynamically set the address for the port), Address_Of (the specified port), and Assign (one the value of one Read_Only_Port to another).

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

8. In reality, the Read_Only_Port class is a metaclass. Users of the Read_Only_Port class must supply:

   - a width (i.e., a non-zero, positive integer value) which will be used to set a fixed width (in bits) for all instances of the Read_Only_Port class,
   - an integer class which will be used to contain the values read by the port, and
   - a system address class, instances of which will be used to dynamically assign a given Read_Only_Port to a specific system address.

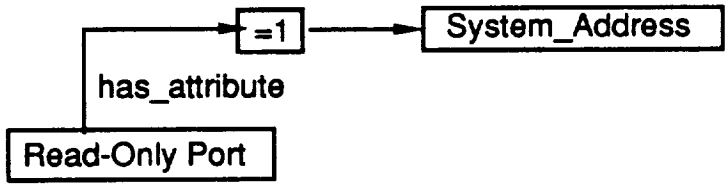
9. The Read_Only_Port class will provide no constants.

10. The Read_Only_Port class will provide an exception: Address_Not_Defined.

2.0 Graphical Representations

2.1 Static Representations

2.1.1 Semantic Networks

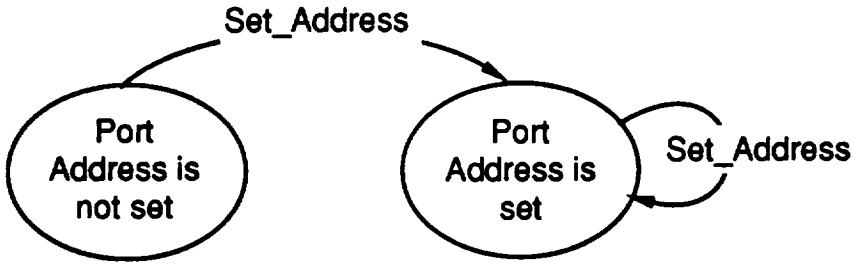


2.1.2 Notes on the Semantic Networks

2.2 Dynamic Representations

2.2.1 State Transition Diagrams

2.2.1.1 State Transition Diagrams for Non-Spontaneous State Changes



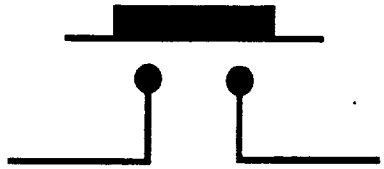
18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

Object and Class Specification

Class: Button

1.0 Precise and Concise Description

1. A "real world" button is made of some hard material (usually plastic and metal) and is used to signal the occurrence of some external event (usually by closing a circuit). In most cases, a button is a two state device (e.g., "pressed" and "not pressed") although it is possible for a button to have more than two states.



2. A button is an "object with life" which is used by an outside source to request service from the system.

3. The required operations for the button are Signal and Press. Press is the operation which connects an instance of this class with the "outside world" (e.g., with a port) so that it knows that a "real world" button has been pressed. Signal is an operation which allows the button to alert a designated object, or system of objects, that it has been "pressed."

4. Buttons have no suffered operations. [However, hardware ("real world") buttons suffer the operations of being pressed and released.]

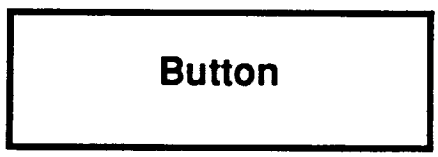
5. The states that the button may be in are "pressed" and "not pressed." Neither of these two states is very persistent.

6. There are no constants or exceptions associated with the button.

2.0 Graphical Representations

2.1 Static Representations

2.1.1 Semantic Networks



2.1.2 Notes on the Semantic Networks

1. To the outside world, a button is a simple object.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

2.2 Dynamic Representations

2.2.1 State Transition Diagrams

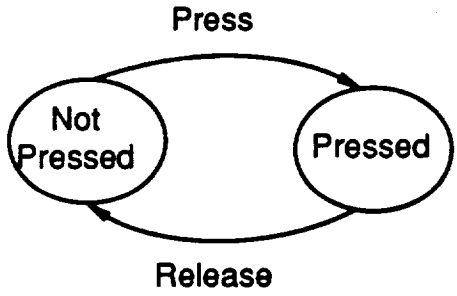
2.2.1.1 State Transition Diagrams for Non-Spontaneous State Changes

1. Not Applicable.

2.2.1.1.1 Notes on State Transition Diagrams for Non-Spontaneous State Changes

1. There are no non-spontaneous state changes for this class.

2.2.1.2 State Transition Diagrams for Spontaneous State Changes



2.2.1.2.1 Notes on State Transition Diagrams for Spontaneous State Changes

1. The "button abstraction" knows when the "real world" button has been pressed via a required operation, i.e., "Press."

2. When the "button abstraction" is made aware that the "real world" button has been pressed, it invokes the "Signal" operation.

3.0 Operations

3.1 Required Operations

Operation	Method
• Signal	• Alerts the button's client that the button has been pressed and returns the buttons identification.
• Press	• Alerts the button that it has been "pressed"

3.2 Suffered Operations

1. Buttons have no suffered operations. [However, hardware ("real world") buttons suffer the operations of being pressed and released.]

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

4.0 State Information

1. The states that the button may be in are pressed and not pressed.

5.0 Constants and Exceptions

5.1 Constants

1. This class will neither provide or require any constants.

5.2 Exceptions

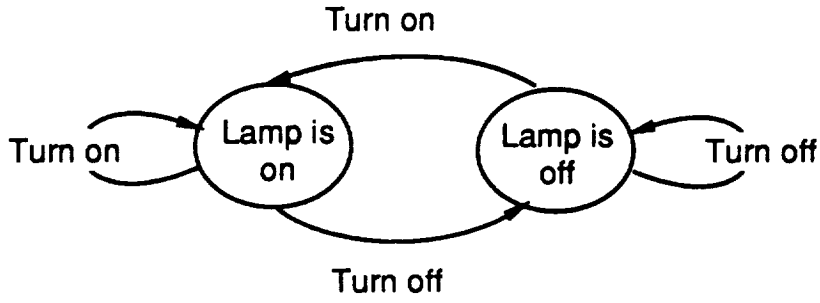
1. This class will neither provide or require any exceptions.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

2.2 Dynamic Representations

2.2.1 State Transition Diagrams

2.2.1.1 State Transition Diagrams for Non-Spontaneous State Changes



2.2.1.1.1 Notes on State Transition Diagrams for Non-Spontaneous State Changes

1. Note that the operation Turn_On has no effect if the lamp is already on and Turn_Off has no effect if a lamp is already off.

2. The Is_On selector operation can be used to determine the state of a given instance of this class.

3.0 Operations

3.1 Required Operations

Operation	Method
• Turn_On	• Connects the lamp abstraction with the means of turning the physical lamp on.
• Turn_Off	• Connects the lamp abstraction with the means of turning the physical lamp off.

3.2 Suffered Operations

Operation	Method
• Turn_On	• Turn the lamp on.
• Turn_Off	• Turn the lamp off.
• Assign	• Assign the state of one instance of this class to another instance of the same class
• Is_On	• Returns true if a given instance of this class is in the "on" state.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

BERARD

4

4.0 State Information

1. The states that the lamp may be in are "on" and "off."

5.0 Constants and Exceptions

5.1 Constants

1. This class will provide no constants.

5.2 Exceptions

1. This class will provide no exceptions.

18620 Mateney Road, Germantown, MD 20874
(301) 353-9652

End Date Aug 20, 1991