INFORMATION INTEGRATION FOR CONCURRENT ENGINEERING (IICE)
IDEF4 OBJECT-ORIENTED DESIGN METHOD REPORT

DRAFT - January 1995

KNOWLEDGE BASED SYSTEMS, INC.
ONE KBSI PLACE
1408 UNIVERSITY DRIVE EAST
COLLEGE STATION TX 77840-2335

HUMAN RESOURCES DIRECTORATE
LOGISTICS RESEARCH DIVISION

AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6573

INFORMATION INTEGRATION FOR CONCURRENT ENGINEERING (IICE)
IDEF4 OBJECT-ORIENTED DESIGN METHOD REPORT

Version 2.0

DRAFT - January 1995

Knowledge Based Systems, Inc.
One KBSI Place
1408 University Drive East
College Station TX 77840-2335

Prepared for:
Armstrong Laboratory
Human Resources Group
WPAFB OH 45433-6573

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This document provides a method overview, practice and use description, and language reference for version 2.0 of the IDEF4 Object-oriented Design Method developed under the Information Integration for Concurrent Engineering (IICE) project, contract # F33615-90-C-0012, funded by Armstrong Laboratory, Logistics Research Division, Wright-Patterson Air Force Base, Ohio, under the technical direction of United States Air Force Captain JoAnn Sartor. The prime contractor for IICE is Knowledge Based Systems, Inc. (KBSI), College Station, Texas. Dr. Paula S. deWitte is IICE Project Manager at KBSI. Dr. Richard J. Mayer is Principal Investigator.

KBSI acknowledges the technical input (Knowledge Based Systems Laboratory, 1991) to this document made by previous work under the Integrated Information Systems Evolutionary Environment (IISEE) project associated with the Knowledge Based Systems Laboratory, Department of Industrial Engineering, Texas A&M University.

FOREWORD

The Department of Defense (DoD) has long recognized the opportunity for significant technological, economic, and strategic benefits attainable through the effective capture, control, and management of information and knowledge resources. Like manpower, materials, and machines, information and knowledge assets are recognized as vital resources that can be leveraged to achieve a competitive advantage. The Air Force Information Integration for Concurrent Engineering (IICE) program, sponsored by the Armstrong Laboratory's Logistic Research Division, was established as part of a commitment to further the development of technologies that will enable full utilization of these resources.

The IICE program was chartered with developing the theoretical foundations, methods, and tools to successfully evolve toward an information-integrated enterprise. These technologies are designed to leverage information and knowledge resources as the *key* enablers for high quality systems that achieve better performance in terms of both life cycle cost and efficiency. The subject of this report, the IDEF4 method, is one of a family of methods that collectively constitute a technology for leveraging available information and knowledge assets. The name IDEF originates from the Air Force program for Integrated Computer-Aided Manufacturing (ICAM) from which the first ICAM Definition, or IDEF, methods emerged. It was in recognition of this foundational work, and in support of an overall strategy to provide a family of mutually-supportive methods for enterprise integration, that continued development of IDEF technology was undertaken. More recently, with the expanded focus and use of IDEF methods as part of Concurrent Engineering, Total Quality Management (TQM), and business re-engineering initiatives, the IDEF acronym has been re-cast, referring now to an integrated family of Integration Definition methods. Before discussing the development strategy for providing an integrated family of IDEF methods, however, the following paragraphs will briefly describe what constitutes a method.

Method Anatomy

A method is an organized, single-purpose discipline or practice (Coleman, 1989). A method may have a formal theoretic foundation, although this is not a requirement. Generally, methods evolve as a distillation of the *best-practice* experience in a particular domain of cognitive or physical activity. The term methodology has at least two common usages. The first usage is in reference to a class of similar methods. According to this usage, one may, for example, hear reference to the *function modeling methodology* when discussing methods such as IDEFØ[1] and LDFD.[2] In the second common usage, the term methodology is used to refer to *a collection of methods and tools, the use of which is governed by a process superimposed on the whole* (Coleman, 1989). Thus, it is common to hear the criticism that a tool (or method) has no underlying methodology when the tool (or method) has a graphical language but no underlying

[1] ICAM Definition method for Function Modeling.

[2] Logical Data Flow Diagramming method.

procedure for the appropriate application of the language or use of the resulting models. The term tool is used to refer to a software system designed to support the application of a method.

Although a method may be informally thought of as a procedure for performing a task and, perhaps, a representational notation, it can be more formally described as consisting of three components as illustrated in Figure 1. Each method has (1) a definition, (2) a discipline, and (3) many uses. The definition specifies the basic intuitions and motivation behind the method, the concepts involved, and the theory of its operation. The discipline includes the procedure by which the method is applied and the method's language, or syntax. The procedure associated with the method discipline provides the practitioner with a reliable process for consistently achieving good results. The method syntax eliminates ambiguity among those involved in the development of complex engineering products. Many system analysis and engineering methods use a graphical syntax to provide visualization of collected data in such a way that key information can be easily extracted.[3] The third element of the method anatomy, the *use* component, focuses on the context-specific application of the method.

[3] Graphical facilities provided by a method language serve not only to document the analysis or design process undertaken, but more importantly, to highlight important decisions or relationships that must be considered during method application. The uniformities to which an expert has become, through experience, attuned are thus formally encoded in visualizations that emulate expert sensitivities.

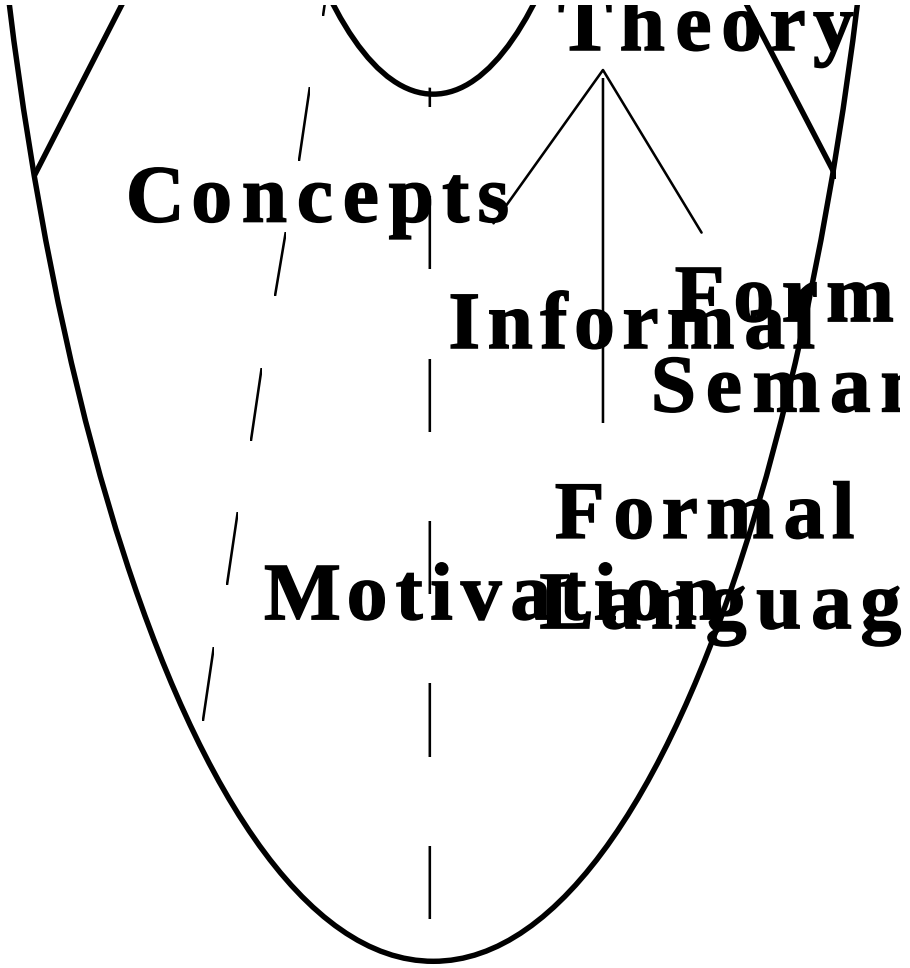


Figure 1
Anatomy of a Method

Ultimately, methods are designed to facilitate a scientific approach to problem solving. This goal is accomplished by first helping one understand the important objects, relations, and constraints that must be discovered, considered, or decided on. Second, scientific problem solving occurs by guiding the method practitioner through a disciplined approach that is consistent with good-practice experience and leads toward the desired result. Formal methods, then, are specifically designed to raise the performance level (quality and productivity) of the novice practitioner to a level comparable with that of an expert (Mayer, 1987).

Family of Methods

As Mr. John Zachman, in his seminal work on information systems architecture, observed:

> ...there is not an architecture, but a set of architectural representations. One is not right and another wrong. The architectures are different. They are additive, complementary. There are reasons for electing to expend the resources for developing each architectural representation. And, there are risks associated with not developing any one of the architectural representations.

The consistent, reliable creation of correct architectural representations, whether they be artificial approximations of a system (models) or purely descriptive representations, requires the use of a guiding method. These observations underscore the need for many «architectural representations,» and, correspondingly, many methods.

Methods, and their associated architectural representations, focus on a limited set of system characteristics and explicitly ignore those that are not directly pertinent to the task at hand. Methods were never intended to evaluate and represent every possible state or behavioral characteristic of the system under study. If such a goal were achievable, the exercise would itself constitute building the actual system, thus negating the benefits to be gained through method application (e.g., problem simplification, low cost, rapid evaluation of anticipated performance, etc.).

The search for a single method, or modeling language, to represent all relevant system life cycle and behavioral characteristics, therefore, would necessitate skipping the design process altogether. Similarly, the search for a single method to facilitate conceptualization, system analysis, and design continues to frustrate those making the attempt.

The plethora of special-purpose methods which typically provide few, if any, explicit mechanisms for integration with other methods is equally frustrating. The IDEF family of methods is intended to strike a favorable balance between special-purpose methods whose effective application is limited to specific problem types, and «super methods» which attempt to include all that could ever be needed. This balance is maintained within the IDEF family of methods by providing explicit mechanisms for integrating the results of individual method applications.

Critical method needs identified through previous studies and research and development activities[4] have given rise to renewed effort in IDEF method integration and development activities, with an explicit mandate for compatibility among the family of IDEF methods. Providing for known method needs with a family of IDEF methods was not, however, the principal goal of the methods engineering activity within the IICE program. The primary emphasis for these efforts was directed towards establishing the foundations for an engineering discipline guiding the appropriate selection, use, extension, and creation of methods that support integrated systems development in a cost-effective and reliable manner.

New methods development has struck out where known and obvious method voids existed (rather than reinventing existing methods) with the explicit mission to forge integration links among existing IDEF methods. When applied in a stand-alone fashion, IDEF methods serve to embody knowledge of good practice for the targeted fact collection, analysis, design, or fabrication activity. As with any good method, the IDEF methods are designed to raise the

[4]Of particular note is the Knowledge-Based Integrated Information Systems Engineering (KBIISE) Project conducted at the Massachusetts Institute of Technology (MIT) in 1987 where a collection of highly qualified experts from academic and research organizations, government agencies, computer companies, and other corporations identified method and tool needs for large-scale, heterogeneous, distributed systems integration. See Defense Technical Information Center (DTIC) reports A195851 and A195857.

performance level of novice practitioners by focusing attention on important decisions while masking out irrelevant information and unneeded complexity. Viewed collectively as a toolbox of complementary methods technology, the IDEF family is designed to promote integration of effort in an environment in which global competitiveness has become increasingly dependent upon the effective capture, management, and use of enterprise information and knowledge assets.

EXECUTIVE SUMMARY

The success of corporations depends on their ability to use information and knowledge assets effectively. To leverage these assets, corporations have invested heavily in information technology. However, along with the advantages of using information technology have come new challenges: spiraling maintenance costs, outdated and inflexible data systems, and software development projects that fail to complete on time and under budget. When proper analysis and planning is not performed and these challenges not addressed, corporations cannot get the full value from their investment in information technology. They may also find they are trapped in a cycle of constantly re-inventing a system because it doesn't fulfill the users' requirements, the technology is obsolete by the time the system is delivered, Commercial Off The Shelf (COTS) software is not capitalized, and system design is either poorly documented or not documented at all.

An effective way to break this cycle is through the application of information-integrated system analysis, design, implementation, and maintenance techniques (such as object-oriented design) (Jacobson, 1994) to record and manage the life cycle of objects in a software system development. This provides traceability[5] from fielded software components to source code objects, design objects,[6] requirements, and the objects in the application domain. Object-oriented techniques also ensure that designers and implementors meet the needs of the system users because they consider real-world objects and the usage of those objects in the system.

What is IDEF4?

IDEF4 is an object-oriented design method for developing component-based client/server systems. It has been designed to support smooth transition from the application domain and requirements analysis models to the design and to actual source code generation. It specifies design objects with sufficient detail to enable source code generation. IDEF4 provides a bridge between domain analysis and implementation (See Figure 2).

In the development of IDEF4, close attention has been paid to

- important features of successful object-oriented techniques and modeling conventions,[7]
- legacy systems re-use through object-oriented encapsulation,
- provision of mechanisms for leveraging client/server technology,
- creation of reusable object-oriented designs and software components,

[5] The ability to audit the design artifact's origins and its evolution.

[6] Elements being designed in an IDEF4 project.

[7] Booch, Rumbaugh's OMT, Shlaer/Mellor, Wirths/Brock, Fusion, Coad and Yourdon, Odell and Martin, and Jacobson (OOSE).

- inclusion of Commercial-Off-The-Shelf (COTS) technology in designs,
- reduction of design complexity through design stratification.

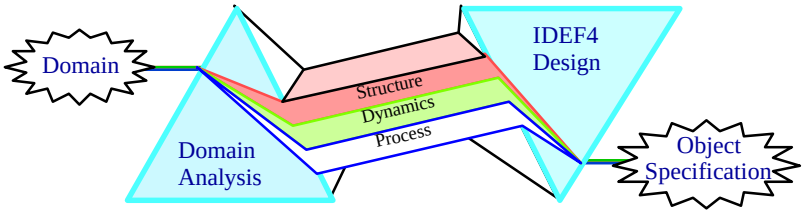


Figure 2
IDEF4 Used Between Domain Analysis and Implementation

IDEF4 supports the design-level description of the external protocols of legacy systems and COTS software. This facilitates the development of object-oriented designs that contain legacy and COTS components, resulting in object-oriented implementations that reuse existing executable software. An object-oriented design philosophy that stresses the separation of external and internal aspects of the design of an object leads to greater design success because it allows for the re-use of design components, concurrent design, design modularity, and deferred decision making.

The IDEF4 method multi-dimensional approach to object-oriented software system design consists of the following items:

- design layers (system-level, application-level, and low-level design),
- artifact design status (application domain, in transition, software domain),
- design models (static, dynamic, and behavior) and the design rationale component, and
- design features ranging from general to specific enabling deferred decision making.

IDEF4 Layers

IDEF4 users design in three distinct layers: (1) system design, (2) application design, and (3) low-level design. This three layered organization reduces the complexity of the design. The system design layer ensures connectivity to other systems in the design context. The application layer depicts the interfaces between the components of the system being designed. These components include commercial applications, previously designed and implemented applications, and applications to be designed. The low-level design layer represents the foundation objects of the system.

IDEF4 Artifact Status

IDEF4 distinguishes between IDEF4 artifacts newly created from the application domain, artifacts in transition to design specification, and artifacts that have been specified that can be applied to create the design specification. Any design artifact in IDEF4 can be marked as domain, transition, or complete. This allows practitioners and reviewers to track the progress of the design toward completion.

IDEF4 Design Models

IDEF4 uses three design models and a design rationale component. The Static Model (SM) defines time-invariant relations between objects (for example, inheritance). The Dynamic Model (DM) specifies the communication between objects and the state transitions of objects. The Behavior Model (BM) defines the relations between the respective behaviors of objects. The design rationale component provides a top-down representation of the system, giving a broad view that encompasses the three design models and documents the rationale for major design evolutions.

Design Features

IDEF4 provides a broad range of design features – from generic to specific. This range enables deferred decision making by allowing the designer to first capture design features in general terms and later to refine them. This significantly reduces the burden on designers by allowing them to immediately capture new design concepts with IDEF4 design features, even if these design concepts have not yet been explored in detail.

Why Use IDEF4?

It is important to use an object-oriented design method to ensure consistency of object-oriented designs. The method used should also ensure design quality by supporting the following activities:

- promoting best practice,
- measuring performance of the individual designers,
- being scaleable across small and large projects,
- being easy to use,
- assessing the progress of the design during each stage,
- working with a broad range of object-oriented system types, and
- being compatible with methods and tools addressing other phases of the life cycle development process.

IDEF4 lets designers solve problems using real-world concepts rather than computer concepts so the resultant software is easier to understand, reuse, and maintain. Focusing on the application domain also ensures that the needs of the system users are better met.

It is not enough to reuse software; most of the intellectual activity in object-oriented development is in object-oriented design, and therefore it is imperative to ensure that design artifacts are also understandable, reusable, and maintainable. Easily understandable, up to date design artifacts that correspond to implementation software greatly improves software maintenance, making it easier to extend systems.

The greatest potential leverage of intellectual investment is object-oriented reuse of executable software. Legacy software can be rendered reusable through object encapsulation, effectively extending its useful life span. The use of commercial software in new systems leverages the intellectual investment of others. Design and implementation of object-oriented software components allows rapid reuse by new systems.

IDEF4 object-oriented technology creates solutions based on real-world concepts and creates system designs focused on quality and maintainability, which result in substantial savings in maintenance costs (Taylor, 1990). Implementation costs must be placed in the perspective of the overall software life cycle. Most of the software costs are incurred after implementation. In fact, it has been estimated that as much as 70 percent to 80 percent of software life cycle costs occur in maintenance. The cost of correcting system defects increases exponentially throughout the software life cycle (Figure 3). IDEF4 designs result in quality software by focusing problem solving on the application domain to satisfy the needs of the customer. The method is designed to help users produce designs, by enforcing design artifact traceability to the application domain and through the design rationale component, that result in low maintenance, extensible software.

Design flaws that surface during operation are extremely expensive to correct (Korson, 1986). Correcting defects in operational software is similar to recalling an entire line of automobiles to correct a design flaw. But rather than having a few hundred or thousand copies to deal with, there may be hundreds of thousands or millions of instances of the software in use. Object-oriented design and development practice have demonstrated the ability to produce software that exhibits desirable life cycle qualities such as modularity, maintainability, and reusability. This is because object-oriented design focuses attention on the application domain, allowing strategic design decisions to be based on user needs.

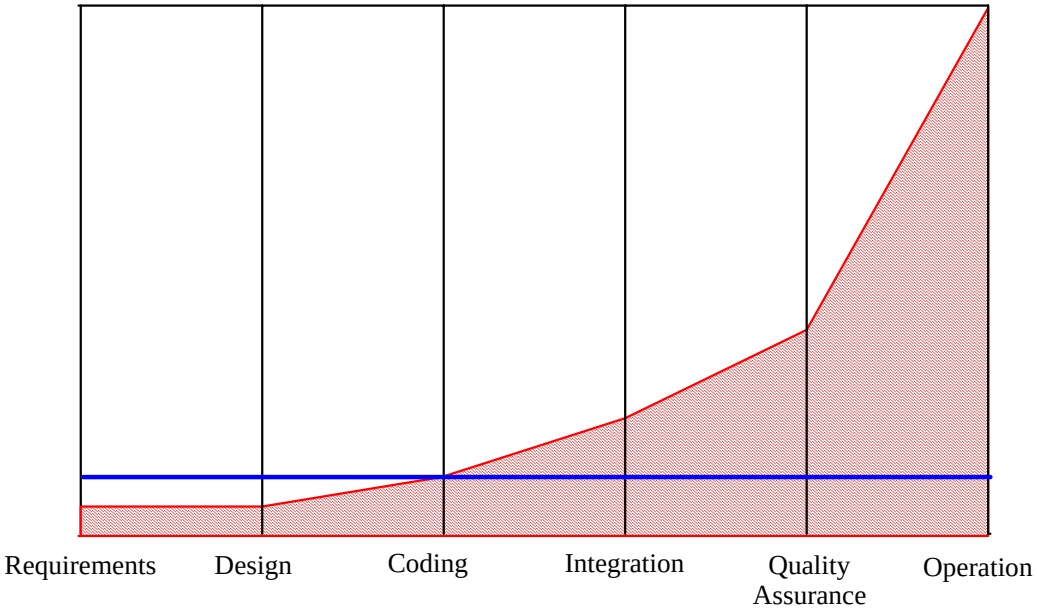


Figure 3
Rising Cost of Correcting Software Errors

Further reductions in maintenance costs can be made by using pre-packaged software components – because they are pre-tested, one need not perform a unit test on each component. Use of component software shifts the emphasis of maintenance to component suppliers and components can therefore be tested independently. The volume of new code needed to implement component-based systems is reduced. Integrating client/server technologies in the design phase results in highly flexible implementations that directly support the work flow through core business processes. This results in increased process effectiveness and simplified re-engineering of systems to support business practices. IDEF4's object-oriented models facilitate communication with customers, visualization of the problem, and reduction in complexity.

How Does IDEF4 Work?

IDEF4 uses an object-oriented design method or procedure that is very similar to Rumbaugh's Object Method Technique (Rumbaugh, 1991) and Schlaer/Mellor's Object-Oriented Analysis and Design (OOA/OOD) technique (Schlaer, 1988, 1991). However, there are some crucial differences: (1) IDEF4 is specifically designed to be compatible with other IDEF methods, (2) IDEF4 allows one to track the status of design artifacts from domain object through transition to design specification, and (3) IDEF4 includes a design rationale component. These extra dimensions are shown in Figure 4; the edges of the box show the progression of the design from start to finish elaborating each of these dimensions.

Figure 4
Dimensions of IDEF4 Design Objects

In IDEF4, a design starts with the analysis of requirements and takes as input the domain objects. These domain objects are encoded in their equivalent IDEF4 form and marked as domain objects. As computational objects are developed for these objects, they are marked as «transitional» and finally as «completed.» The level of completion of an IDEF4 design is determined by setting measures based on the status, level, and model dimensions of individual artifacts in the design.

The system-level design starts once the «raw material» (domain) objects have been collected. This develops the design context, ensures connectivity to legacy systems, and identifies the applications that must be built to satisfy the requirements. Static, dynamic, behavioral, and rationale models are built for the objects at the system level. These specifications become the requirements on the application level – the next level of design. The application level design identifies and specifies all of the software components (partitions) needed in the design. Static models, dynamic models, behavioral models, and the rationale component are built for the objects at the application level. These specifications become the requirements on the next level of design – the low-level design. Static Models, Dynamic Models, Behavioral Models, and the Design Rationale component are built for the low-level design objects. Sub-layers may be built within each layer to reduce complexity.

IDEF4 is a iterative procedure involving partitioning, classification/specification, assembly, simulation, and re-partitioning activities (Figure 5). First the design is partitioned into objects, each of which is either classified against existing objects or for which an external

specification is developed. The external specification enables the internal specification of the object to be delegated and performed concurrently. After classification/specification, the interfaces between the objects are specified in the assembly activity (i.e., static, dynamic, and behavioral models detailing different aspects of the interaction between objects are developed). While the models are developed, it is important to simulate use scenarios or cases (Jacobsen, 1994) between objects to uncover design flaws. Based on these flaws the designer can then re-arrange the existing models and simulate them until the designer is satisfied.

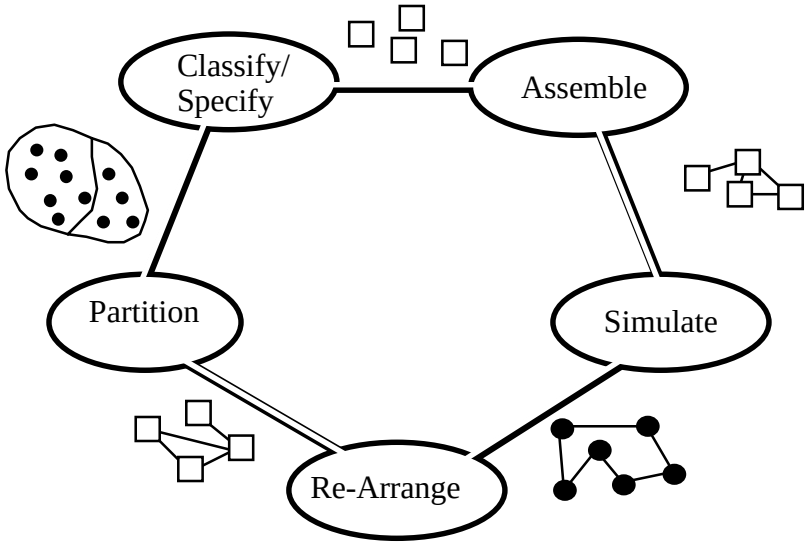


Figure 5
IDEF4 Design Activities

Purpose and Organization of this Document

The purpose of this document is to provide information about the IDEF4 object-oriented design method. The method report meets the following results:

(1) Describe the IDEF4 object-oriented method,

(2) Enable systems analysts/designers to understand the process of producing object-oriented designs using IDEF4, and

(3) Enable software engineers to read and interpret IDEF4 designs.

The scope of this document encompasses the following items:

(1) An introduction to the IDEF4 method,

(2) An overview of the IDEF4 model structure and graphical syntax,

(3) A description of the IDEF4 object-oriented system-design development procedure, and

(4) Examples of the use of IDEF4 derived from its application on object-oriented design case studies.

Within this scope, the IDEF4 Method Report is targeted for the following audience:

(1) IT Managers,

(2) Software system architects,

(3) Project managers,

(4) Systems analysts/designers, and

(5) Implementors of object-oriented designs.

Summary

IDEF4 is a method that has been developed to (1) apply object-oriented design techniques within the context of the IDEF family of methods, (2) separate an object's external and internal design specifications, (3) design systems which can interface to legacy systems and commercial systems, (4) employ and update the design during system use and maintenance, (5) reuse design objects in other designs, and (6) specify object-oriented, distributed computing environments. Reuse of design artifacts, software objects, and executable components is facilitated by syntactic support for encapsulating components. The maintainability of design and software components is enhanced by emphasis on documentation and artifact traceability.

INTRODUCTION TO IDEF4

The IDEF4 method was developed to reduce the risk in object-oriented system development by assuring understanding, reuse, and maintainability. Understanding of IDEF4 is facilitated through the use of a clear, concise graphical syntax that supports reduction of design complexity by supporting modular design. The design is divided into static, dynamic, and behavioral models. Reuse of design and software components is enhanced by encapsulation (Mayer, 1987) of design, implementation, and executable components. Maintainability of design and software components is ensured by separation of internal and external object specifications, emphasis on documentation, and traceability of artifacts using design rationale capture.

This chapter describes: (1) the object-oriented concepts used in IDEF4, (2) the IDEF4 procedures, and (3) the organization of IDEF4 designs.

IDEF4 Object-oriented Concepts

One of the greatest barriers to object technology is the specialized vocabulary that has evolved around it (Taylor, 1993). Similarly, there is a great deal of confusion centered around the use of different terminologies by different object-oriented languages. Because terminological confusion leads to conceptual confusion and vice versa, it is imperative to define IDEF4's object-oriented terminology before proceeding.

Domains

IDEF4 projects are implemented in a domain. A domain can be seen as the scope of the system being developed. During system design, the software is transitioned between three domains: the application domain, the design domain, and the implementation domain.

The application domain is the domain the application is being designed for. This domain deals with the end users. Design artifacts, which are any «*things*» that will be used in the design, are identified in this domain. Once the application domain has been defined, design techniques are applied to transition to the design domain. The design domain is where the decision on *how* the system will be implemented is made. For example, the design domain for a project using the IDEF4 methodology is the IDEF4 method. All of the design artifacts will be depicted using IDEF4 syntax. After the design technique has been decided, it is transitioned to the implementation domain through code generation. The implementation domain is where the implementation of software occurs based on the design artifacts defined in the design domain. The actual software is implemented and fielded, causing it to transition back to the application domain where end users' evaluations, comments, and request for changes will start the system development cycle over again. When making bug fixes in software, developers may sometimes skip the design domain and go straight from the application domain to the implementation domain. This makes the actual system design useless because changes have been made to the software that are not captured. The software therefore cannot be reused. It is much more efficient to develop software using all domains.

A domain contains objects and messages. These are the two most important object-oriented concepts, and they are the building blocks of the IDEF4 methodology.

Features, Artifacts, and Objects

Since «object» is a special kind of feature in IDEF4, discussing features as an introduction to the concept of objects will result in a clearer understanding of objects. Feature is a catch-all category for concepts that the engineer wants to capture for the software design. These concepts include physical objects, descriptions of physical objects, COTS software the engineer wants the software being designed to interface with, and information about the required behavior of the elements in the software. Once these features are documented, the designer can then begin classifying them as objects, relations, links, methods, attributes, and events. Figure 6 shows a Bank object with many of these kinds of features. In IDEF4, the symbols O, R, L, M, A, and E are used to denote objects, relations, links, methods, attributes, and events, respectively. Users of IDEF4 can use these feature types to defer more detailed design decisions until later in the design. Feature types allow features to be initially characterized in general terms which can later, as the design progresses, be revisited and evolved to a more specific definition. For example, a designer might first specify *Name*, a characteristic of an object «Bank,» as a feature. As the design evolves, the designer can specialize the definition of *Name* to be an attribute that describes the object «Bank.»

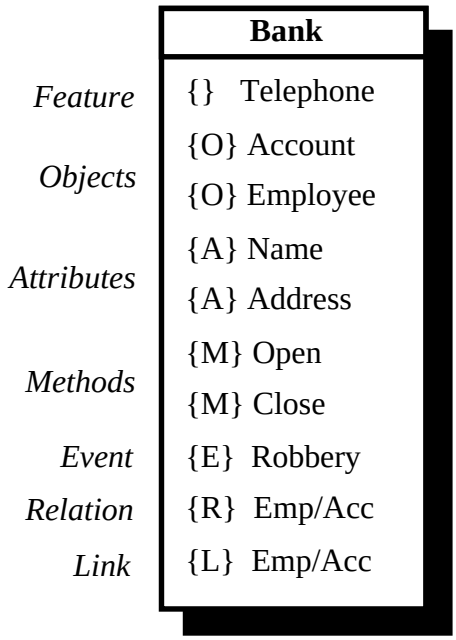


Figure 6
The Bank Object

Again, our explanation of objects in terms of features is because objects are features; i.e., feature is a broad category and object is one of its specializations. Objects themselves can also encapsulate features, some of which may be specialized to objects, which will result in objects encapsulating objects, as is the case in the banking example shown in Figure 6.

In this example, «Bank» is referred to as an «object.» In IDEF4, the term *object* refers to information which serves as a specification for the implementation. Objects have «behavior» which describes how the object will act in the system, and «states» which describe the value of object characteristics. Objects are identified in the application domain. They fall into five categories:

(1) physical – a physical object such as a tree, a car, a plane, a person;

(2) role – an object with specific behavior that establishes its identity, such as a nurse, fireman, or professor;

(3) event – an object representing an occurrence such as a meeting or a delivery;

(4) transaction/interaction – the capture as an object of an interaction between objects such as the telephone wire between two telephones transmitting a message; and

(5) specification procedure – an object that represents a set of instructions, such as a car design.

Objects in a domain should be identified according to these categories and according to the order of the previous list (i.e., looking for physical objects first and specification procedures last).

IDEF4 artifacts can exist in four distinct life cycle stages: *application-domain* artifacts, *design* artifacts, *specification* artifacts, and *software* artifacts. The first three life cycle stages reflect the state of the artifact in the design. An application domain artifact is any feature existing in or abstracted directly from the application domain. For example, «passenger airplane» becomes an object in the domain; but the actual element stored about the plane is its behavior in relation to an airline reservation system: the number of seats, plane type, and flight destination (Figure 7). Application domain artifacts are the «raw material» that is input into the design.

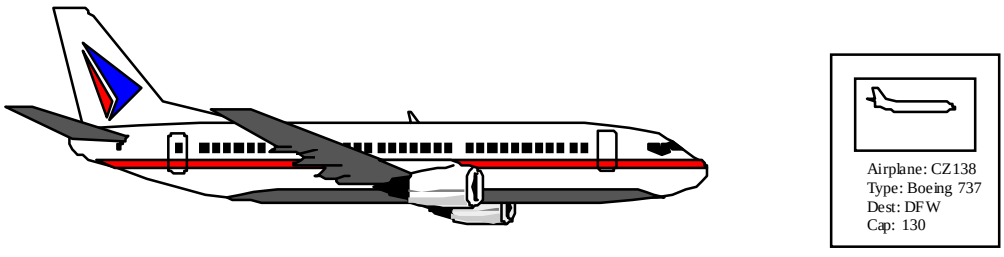


Figure 7
Abstraction of Passenger Aircraft

Once an application domain artifact is worked on in the design it becomes a design artifact. Design artifacts evolve to specification artifacts that are used for generating software.

Design artifacts evolve in a design to satisfy a computational purpose. Design objects contain a collection of related procedures and variables and may be an abstraction of a real-world object (e.g., an aircraft object). Design artifacts can be viewed as «work in process» and are used in the transformation from application domain objects to specification objects. Design artifacts typically start as application domain artifacts and are gradually transformed to specification artifacts which are the final design specification. Specification artifacts can be viewed as «finished goods.»

The «software object» concept has its origins in the need to model real-world objects in computer simulations. A software object is a software package that contains a collection of related procedures and data. In the object-oriented vernacular, the procedures are referred to as *methods* and the data elements are called *attributes* or *variables*.

IDEF4 uses the notation «®» to indicate that artifacts refer to the application domain, «Δ» to indicate that the artifact is in transition (i.e., design artifact) and «©» to indicate a completed design (i.e., a specification artifact).

To illustrate the concept of a design artifact, consider how one might represent an airplane in a simulation. The airplane can exhibit a number of behaviors, such as flying from one location to another and loading and unloading passengers. The computational representation must model the airliner's behavior, for example, its ability to fly at certain speeds, its ability to seat passengers, and its ability to fly from location X to location Y. The state will also be modeled (e.g., number of passengers, location, altitude, orientation, and velocity). In a design object, you would describe the airplane's behaviors as methods, and the airplane's state as attributes that would be effected by the methods. Figure 8 shows the process of abstracting a passenger aircraft, encoding it as «raw material» for IDEF4, evolving it toward a computational specification as «work in process,» marking it as a completed specification, and finally implementing it in a program as an object that represents the airliner in the airline reservation system.

Figure 8
Design Artifact Used For Design Evolution

Object Instance

Objects can be *object instances*, *object classes*, and *object partitions*. Object instances are the individual things encountered in the application domain. For example, (Figure 9) the employee named «Toonsie» and the car with Texas License «BYJ14T» are examples of object instances.

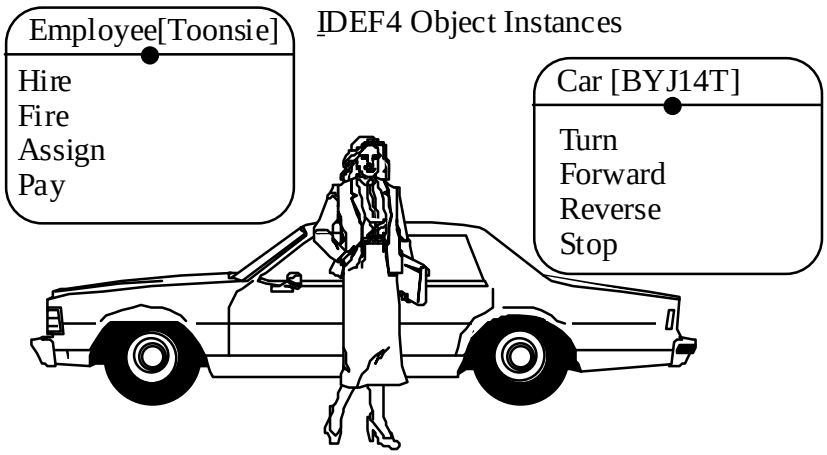


Figure 9
IDEF4 Object Instances

Figure 9 shows IDEF4 Object Instances for Toonsie and her car. These instances have distinct behaviors that can be performed by the objects. For example, Toonsie is an employee and can be hired, fired, assigned, and paid; the car can turn, go forward, reverse, and stop. Because groups

of object instances can be very similar, it is advantageous to define characteristics and behavior for groups of similar object instances. Such a group is known as an Object Class.

Classes

Classes are generalizations about objects and are used to manage complexity by taking advantage of similarities in object instances and grouping them under a class or category. It is more efficient to attach common characteristics and behavioral information of object instances to an object class, rather than to each individual object instance.

The examples shown so far involve only instances of particular kinds of objects. It is much more common to need more than one object instance of each kind. For example, a company will have many employees who have many cars. We call such a grouping of objects with similar behavior an object class. The object class concept is a convenient way to solve the problem of redundantly defining the same behavior for similar object instances. In software systems, the object class acts as a template for the variables and methods that need to be included in every object instance. Object instances contain only the particular values of their variables and share the methods stored in the class object. A software object class can be viewed as a factory for manufacturing software object instances.

Object classes are useful in application domains. For example, object instances *John*, *Susan*, »*Bill*, and *Mary* fall into the object class *Employee* because they are all people employed by a company (Figure 10). An object class represents a set of object instances that have the same *features*; i.e., they have the same characteristics and conform to the same rules, policies, and procedures.

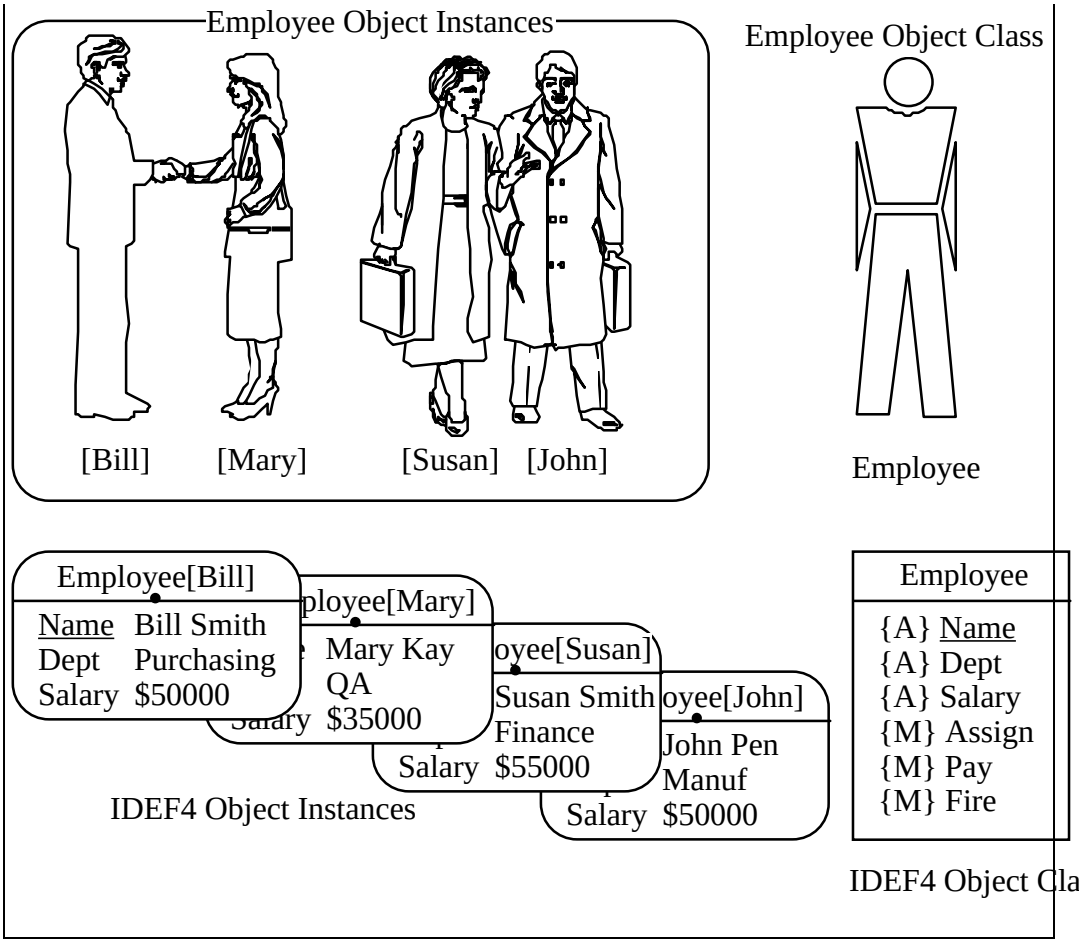


Figure 10
IDEF4 Object Instances and Object Classes

The classes in an object-oriented system define the types of objects that exist within the system. Each class contains a set of traits that define the characteristics and the rules, procedures, and policies that instances of the class must abide by. The set of traits consists of attributes, methods, embedded classes, and relationships. The attributes are used by the object instances contained within a class to store their state. The methods characterize the behavior of object instances in a class. The classes can also be organized into subclasses and superclasses.

Subclass/Superclass

The term *subclass* captures the concept of grouping particular instances of a class into an even more specialized class. For example, a class may have object instances of employees. Managers are employees that can delegate responsibilities to other employees. Not all employees are managers and, therefore a manager is considered a special kind of employee (Figure 11). We say that manager is a *subclass* of employee. The term superclass refers to the class from which a subclass is derived; e.g., the object class *Employee* is a superclass of the object class *Manager*.

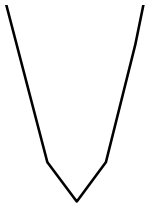


Figure 11
Inheritance Relating Subclass and Superclass

Classes are the major syntactic construct in IDEF4, as in all object-oriented methods. Partitions are another construct used in IDEF4, and they are similar to classes in that they group objects.

Partitions

A *partition object* contains objects and relations.[8] A partition object is essentially a self-contained IDEF4 object-oriented model that itself may contain submodels. Partitions are similar to software modules, packages, components, files, or a collection of cooperating software objects. In the application domain, partition objects contain objects that give the partition its essential character; objects in a partition are only accessible through the partition object.

Semiconductor chips provide a good example of a partition object (Figure 12). The external logic of the chip is specified by a very complex internal structure consisting of a network of transistors and resistors. Users of the chip only need to know the external logic and are not concerned with the individual components used to make up the chip. This is how partition objects work; the substructure (compared to the internal logic of the chip) dictates how the partition behaves. The partition object itself provides an external interface so the user does not have to be concerned with the objects in the substructure.

[8] The partition construct has its origins in the container objects of the DKMS program (Keen, 1991), packages in CLOS, imbedded objects in C++, Domains in Schlaer/Mellor, and Use Cases in Jacobsen.

Figure 12
Semiconductor Chip is made of Transistors

Partitions are also used for incorporating legacy and COTS software within a design (encapsulation) by specifying an object-oriented external interface for the object that receives and relays messages to and from the contained executable software. For example, a manufacturer has software for controlling a numerical control milling machine. The software interprets a file containing instructions that specify the tool path and issues commands to the mill. The manufacturer can reuse this software in a new computer integrated manufacturing system by wrapping code around the legacy software (encapsulating the legacy code) to perform the necessary communications with the new system. This wrapping code would be specified using an IDEF4 partition.

Partitions allow the designer to carve up the design into self-contained subdesigns that may be worked on by many teams (Figure 13). The partition concept allows the designer to partition the system hierarchically in terms of communicating objects of different levels of abstraction.

Figure 13
The Partitions Manage Design Complexity

Object classes and partitions provide a way to design a system with managed complexity. Partitions group objects into a substructure that can be reused in other models. Object classes group objects by their behavior so that the behavior information is only stored once; the behavior information can then be referenced by the object instances. Object States are necessary for describing object behavior. Attributes are an implementation choice for representing an object's state.

Attributes

Attributes are an implementation choice on how to represent an object's state. Each object instance has attributes or characteristics. For example, Mary is 5'8" tall, her social security number is 555-77-4444, she owns a car and works at ACME (Figure 14).

Figure 14
Information Kept By Employer about Employee and Car

The attributes hold information about the state of an object and are necessary for implementing behavior. Object Instances know the rules, policies, and procedures of the domain in which they exist; e.g., Mary knows that she must be at work at ACME before 7 a.m., she is attuned to traffic laws that govern her drive to work, she knows that she must park her car in the employee lot, and she is responsible for maintaining inventory levels at ACME. All of these behaviors need to access and manipulate information about the state of the objects that they pertain to. The ACME corporation needs to keep information about *Mary* as an employee. They also need to keep information about her car to ensure that the employee parking lot is only used by employees. Figure 14 shows how this object instance information is maintained in an IDEF4 model. The characteristics that have been recorded about Mary and her car are called attributes. The attributes that uniquely identify an object instance are called identifier attributes. Identifying attributes are shown underlined in IDEF4. Attributes that point to other objects, for example, the attribute car in the object instance Mary are known as referential attributes. Attributes that describe an object instance but do not uniquely identify it, such as height, weight, and eye color, are known as descriptive attributes.

Object States

Object states represent situations or conditions of an object instance that are meaningful in the design. For example, in one system it may be appropriate to recognize the states cold, hot, and boiling for a pot of water, whereas in another system, the actual temperature might be more appropriate. The water heater shown in Figure 15 has three states: (1) cold, (2) normal, and (3) hot. When the water heater is in the cold state, the heater is turned to full, when the heater is in the normal state, the heater is turned to half, and when it is in the hot state, the heater is switched off.

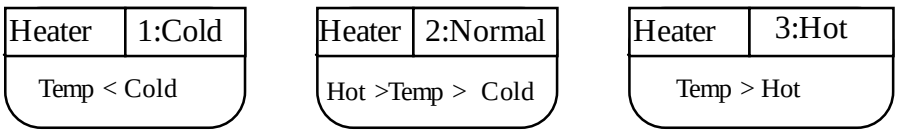


Figure 15
States For Heater

The level of abstraction of object states is dependent on the representational needs of the system in which they must exist.

Objects are multidimensional entities that store many different types of information. By creating partitions and classes, and thereby simplifying the diagrams, IDEF4 has made the management of objects more accessible. The designer can then look at objects at different levels of complexity. The next section will discuss how the designer can define interaction between these objects and between different levels of complexity.

Method

A *method* is an implementation of behavior (i.e., a set of instructions according to which the object performs some operation). An object's methods are a collection of procedures owned by the object that give the object its essential character, dictating how the object will behave within the system being designed. They can be thought of as verbal descriptions of the things an object can do. For example a flight object used in a system design for airline reservations will have procedures for assigning and canceling passengers on the flight (Figure 16). These procedures are associated with the object instances of flight by defining them as *methods* in the flight object class. The *Reserve* and *Cancel* methods accept four parameters: (1) the ID or handle of an authorized travel agent, (2) a password, (3) the seat number, and (4) the passenger's name.

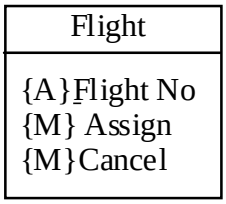


Figure 16
The *Assign* and *Cancel* Methods for Airline Flight Reservation

Behavior specifications (i.e., how the method must be implemented by a software engineer) are part of the Behavior Model. Each method in IDEF4 must have a behavior specification associated with it.

Message and Polymorphism

Objects communicate by sending *messages* to each other. Methods, in turn, tell the object how to implement a message. Each object should know how to implement the messages that are sent to it. If the object can't respond, it should signal an exception (error). A message consists of the name of a method that the object knows how to execute and the object identifier or handle. If the method needs more information to execute, then the message may also supply additional parameters. An object can send a message to one object, many objects, and even to itself. Communication between objects is depicted in IDEF4 by an arc with a chevron pointing in the direction of the communication.[9]

A good example of messaging is an orchestra conductor. An orchestra conductor sends messages to the members of the orchestra by using gestures (Figure 17). These gestures are messages that tell the members of the orchestra to play louder, softer, faster, or slower. The message *Play_Louder* is implemented differently by strings, brass, woodwinds, and percussion players. This aspect of messaging is known as *polymorphism*; i.e., the same message is implemented differently by different objects.

[9] The chevron is used to depict a message or signal being sent. The symbol was derived from client/supplier link used in IDEF4 revision 1.

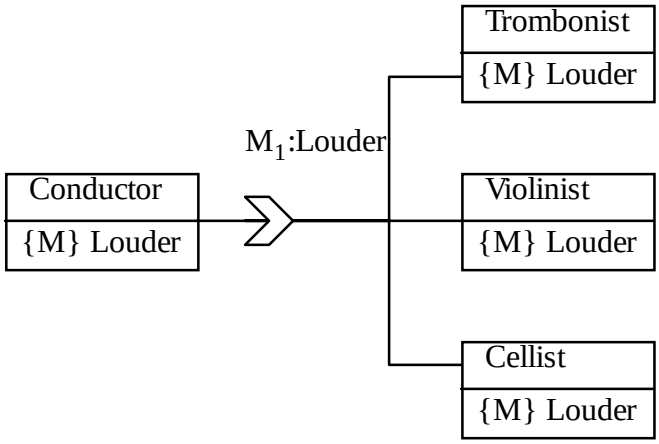


Figure 17
Communication between Conductor and Orchestra

Figure 18 uses an orchestra to describe the relationship between messages, classes, and methods. In an orchestra, the conductor sends the message «Louder» to musicians. Each kind of musician employs a different method to effect the behavior intended by the message. Figure 18 shows the relationship between the message «Louder» and the methods implementing it in different classes.

Figure 18
The Behavior Diagram for methods Implementing «Louder»

The Method Specification contains the message, the applicable class, and the methods employed. The specialization relationship between method boxes indicates whether the relationship is based on external (e) or internal (i) specifications. The difference between the

way musicians play louder on brass instruments and stringed instruments is externally visible, whereas within the family of stringed instruments, the differences are not externally observable, so the differences are internal. These differences point to whether the external specification or internal specification of the method is being specialized.

The *sender* does not have to know exactly how the message is implemented. Often the only thing the sender needs to know is whether the message has caused a transaction to take place or not. This is known as a synchronous message. In IDEF4, messages are synchronous by default. If the sender does not need to know the effects of the message, it is known as an asynchronous message. Asynchronous messages are typically initiated by an *event*.

Event

An event is a signal generated by a method in an object indicating some condition in the object. An event being broadcast by one object does not necessarily mean that there will be any action taken by other objects. For example, the beep of a smoke alarm indicates that smoke has been detected. It does not indicate what action needs to be taken by any agent hearing the beep. In order for the events generated by objects to be meaningful in any system, they need to be associated with messages telling other objects to perform some action. For example, the «disconnect battery» message is associated with the smoke alarm event, causing the object to look up procedures for disconnecting the battery from the smoke alarm to turn the alarm off.[10]

Object Life cycles

In any system, objects exhibit patterns of behavior as they cycle through different states. An event or message will initiate a transition of an object from one state to another. These events and messages are triggered by conditions within the object itself or within objects that communicate with the object. For example, in a hot water system, the states cold, normal, and hot are important. The diagram in Figure 19 describes how the system cycles between the three states: (1) cold, (2) normal, and (3) hot. When the water heater is in the cold state, the heater is turned to full, when the heater is in the normal state, the heater is turned to half, and when it is in the hot state, the heater is switched off.

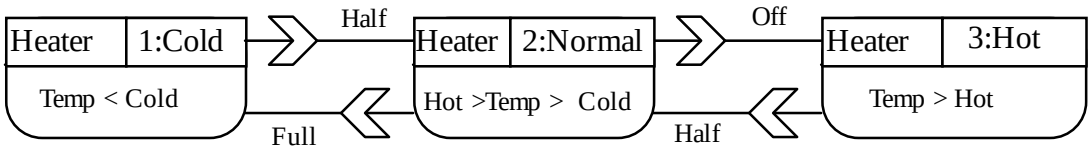


Figure 19
Water Heater Object Life Cycle

[10] N.B. Although in this example the battery is taken out of the smoke alarm to turn it off, this is not an endorsed means for handling this situation.

Client/Server

An object plays the role of a *client* relative to a message if it is the *sender* of that message. An object plays the role of a *server* relative to a message if it is the *recipient* of that message. If an object normally plays the role of initiating contacts with other objects it is known as a client. If it is generally responding to messages sent to it, then it is a server.

Figure 20
Objects Communicating With Messages

Figure 20 describes the communication between a starting judge communicating with sprinters at the start position. The starting judge issues the commands «On Your Marks,» «Get Set,» and then fires the starter pistol. If one of the sprinters starts running before the gun is fired, then the judge fires the pistol twice, signaling a false start. The sprinters respond to these events by positioning themselves at the starting line, moving to the set position, and running to the finish line.

E1:Marks->M1:Position

E2:Get_Set->M2:Set

E3:Bang->M3:Run

E4:BangBang->M4:False_Start

Relationships and Roles

Figure 21 shows objects connected together with arcs. These arcs are called relationships and they show associations between objects.

Relationships exist between objects; for example, ACME monitors cars parked in the employee lot. Most employees drive to work. Some employees own more than one car and in the case of married employees, cars may be owned by more than one employee. Figure 21 shows how this relationship is represented in IDEF4.

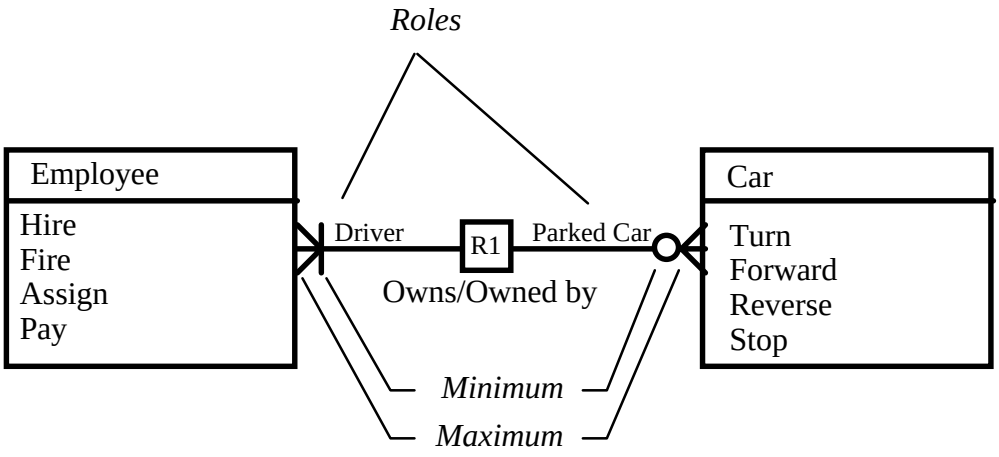


Figure 21
Employee/Car Relation

The relationship is drawn as an arc connecting the Employee and Car classes. The notation on each end of the relation arc denote the cardinality or multiplicity of the relation.[11] This relationship is read as an employee owns zero or more cars and a car parked in the employee lot must be owned by one or more employees.

Relationships have roles which dictate how the objects participate in the relationships. Note the *roles* taken on by employee and her car. The employee takes on the driver role with respect to the car and car takes on the parked car role with respect to the employee. The behavior of the employee and the car are governed by certain policies, procedures, and rules in these roles. The employee must drive the car according to the rules and regulations set up by the Department of Transportation. The car must be parked according to the rules for employee parking at ACME.

Inheritance

A specific type of relationship used in object-oriented technology is inheritance. Inheritance, in fact, is perhaps the most distinguishing characteristic of object-oriented technology. Inheritance is a way of implementing generalization/specialization relationships between object classes. A generalization/specialization relationship occurs when a subclass inherits features from one or more superclass. An example (Figure 22) would be the superclass *Vehicle* which contains the subclasses *Car* and *Truck*. The inheritance relationship is denoted by a triangle[12] on the arc between the subclass(es) and superclass pointing in the direction of the superclass.

[11] IDEF4 uses the same cardinality symbols used in Entity-relationship diagrams because they are easily understood and have been in use for many years in conventional systems analysis. These symbols are also use in Martin and Odell's «Object Oriented Systems Analysis and Design» (Martin, 1992).

[12] The triangle notation is the same as that used in the Object Method Technique (Rumbaugh, 1991).

Figure 22 - Truck/Car Inheritance From Vehicle

Vehicle would store all the methods and attributes that are common to both trucks and cars. If an instance of *Car* was queried to find out the car's mass, *Car* would get the information about mass from *Vehicle*, where it is stored. When subclass/superclass relationships are created, the instances of the superclass are normally partitioned into mutually exclusive, totally exhaustive sets denoted by a blank triangle. In the case of overlapping categories, the triangle is filled in black.

From the real-world point of view, the inheritance phenomenon operates like a specialization relation by dividing objects into more specialized groups. The inheriting class (subclass) is a specialization of the class from which it inherits (superclass). The concept of inheritance allows for the reuse of methods and class features within the inheritance hierarchy. Once the methods and features have been defined in one class, they can be used, through the subclass/superclass relationship, by any other object or class that is similar.

Encapsulation and Information Hiding

Encapsulation and information hiding are two object-oriented concepts that are most easily understood when discussed in terms of interactions between objects.

In the orchestra conductor example, the conductor did not need to know how the messages being sent to the orchestra were implemented by different members. This is known as *information hiding.* Information hiding is used when it is unnecessary to expose implementation details – it explicitly limits access to the internal structure of objects. IDEF4 provides explicit support for information hiding by providing levels of information hiding in classes (Figure 23) and internal and external specifications for all design artifacts. A class has private and public levels of information hiding.

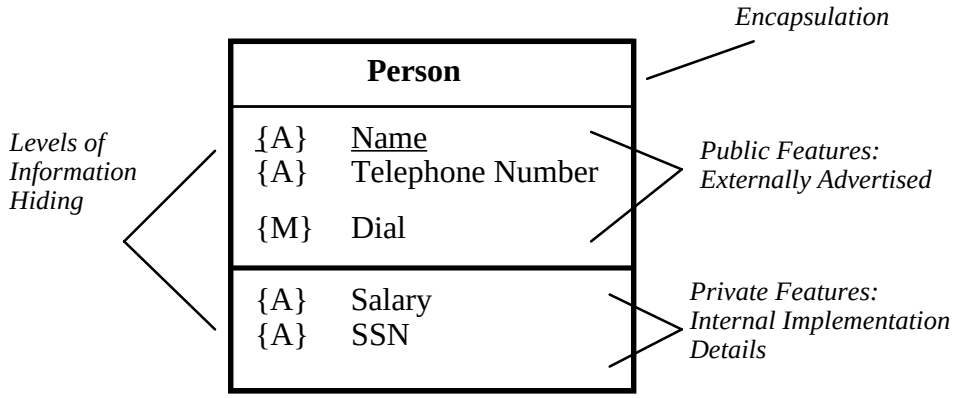


Figure 23
Levels of Encapsulation in IDEF4

Features in each level are protected according to the level information hiding. Private features are intended for use by instances of the class only. Public features are documented, advertised, and supported for external use. In Figure 23, the class person, intended to support automated dialing, encapsulates name, telephone number, salary, social security number, and a dial behavior. The person's name, telephone number, and the capability to automatically dial the person's number are public features, whereas the salary and social security number are not.

Information hiding is made possible by encapsulation using objects. IDEF4 objects encapsulate behavior, data, and objects. Encapsulation can be thought of as a container for information about an object. This container keeps the information in one place, in effect modularizing it. Information hiding is typically used on objects to expose their external interface while hiding internal workings; information hiding controls how much of the inside of the container is visible. For example, an object that needs to use the services of another object need only know what an object does, not how it does it. A person that needs to know the time from another person need not know how that person tells time.

Objects that encapsulate objects are known as partition objects in IDEF4. With partition objects, it is important to have a high degree of cohesion (Yourden, 1979) between objects encapsulated by partitions and a low degree of coupling between partition objects (Figure 24). This maximizes the re-useability and maintainability of objects. In Figure 24, the partitions *Package1*, *Package2*, and *Package3* are loosely coupled, while their embedded objects are tightly coupled.

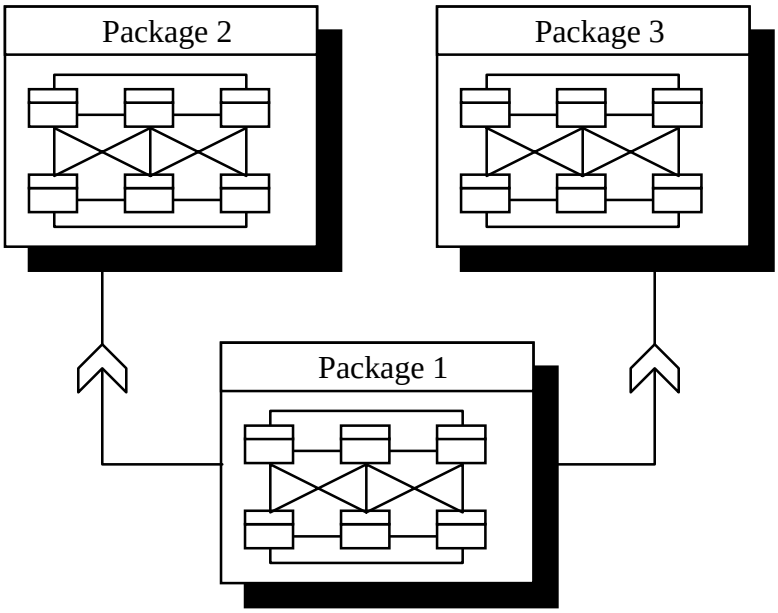


Figure 24
High Internal Object Cohesion Versus Low Partition Coupling

IDEF4 Procedure

IDEF4 is an Object-Oriented Design method providing a multimode approach to transforming application domain information to design specifications (Figure 25). IDEF4 is focused on the design process, relying on IDEF domain analysis methods or other methods to supply models of the application domain. The IDEF4 process outputs design specifications to an implementation process. The major activities in IDEF4 object-oriented design are (1) requirements analysis, (2) system design, (3) application design, and (4) low-level design.

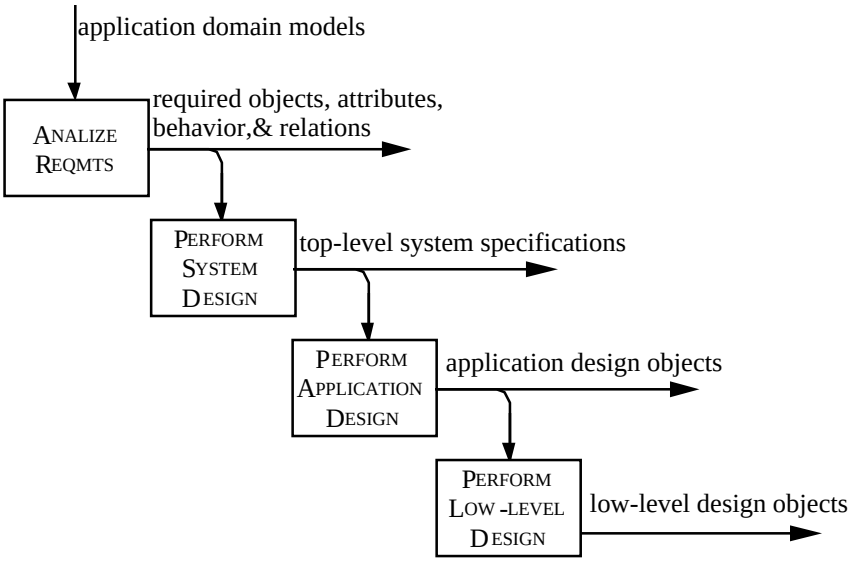


Figure 25

The Modes of IDEF Object-Oriented Design

The requirements analysis activity processes information from domain models and requirements documents in order to establish clear and concise requirements that will serve as a base for the other activities. The requirements analysis activity provides initial objects and use scenarios. The use scenarios map to requirements and are used to validate requirements satisfaction in design and implementation[13]. The system design activity establishes design strategies, divides the design into top-level design partitions, and defines the interfaces with other systems. The application design activity specifies the object-oriented design details in these partitions. The low-level design activity generates language specific implementation specifications.

IDEF4 Organization

IDEF4 design partitions are described in three models:

(1) Static Model (SM),

(2) Dynamic Model (DM), and

(3) Behavior Model (BM).

IDEF4 also includes the Design Rationale, which provides a top-down view of the system and documents design changes.

Each model represents a different cross section of the design (i.e., the static object structure, dynamic object interaction, and behavior). The three design models, accompanied by the Design Rationale component, capture all the information represented in a design project. Each model is supported by a graphical syntax that highlights the design decisions that must be made and their impact on other perspectives of the design. In order to promote ease of use, elements of the IDEF4 method are uniformly represented between models; e.g., a class is depicted as a box in all models.

Static Model

The Static Model specifies the *static* structure of the design objects (i.e. the design schema). It includes information such as the objects that will be needed in the design and instances of those objects, attributes describing the objects, classification of objects into sub- or super-classes, and relationships and roles among the objects. Examples of the information depicted in a static model could include the following items:

(1) *subclass/superclass* relationship of class objects; e.g., an employee is a subclass of the superclass person,

[13] Use scenarios correspond to use cases and use case analysis in Jacobsen's Object-Oriented Systems Engineering (OOSE) (Jacobsen, 1994).

(2) operations that can be performed on an object; e.g., a document can be printed, a person can be hired, an employee can be assigned,

(3) two way relationships; e.g., *employs/employed-by* relationship between class objects *employee* and *employer,* and

(4) attributes of the class object; e.g., the class object *person* may have attributes of *age*, *name*, *weight*, and *height.*

The Static Model includes object specifications and structural connections between objects. These connections include inheritance, subtyping, relationships, and links.

Dynamic Model

The Dynamic Model specifies communication between objects and transitions between states. It contains a graphical depiction of the messages that are relayed between objects, events which cause the object to implement the message, and the resulting transition of the object from one state to another as the object executes the message. Examples of communications that may be specified include the following items:

(1) an object instance «switch» transitioning between the states «on» and «off,»

(2) the «open» event generated by clicking on a file icon and the «launch» message sent to its corresponding application,

(3) the «door-open» event generated by a sensor software component and the «sound-alarm» message sent to the security management component, and

(4) a conductor sending a «play louder» message to members of the orchestra.

The Dynamic Model is specified in client/server diagrams and state diagrams using objects or object states linked by messages and/or events.

Behavior Model

The Behavior Model specifies the implementation of *messages* by *methods* in objects. It contains behavior diagrams that show relationships between methods that implement a message or behavior. The behavior diagram specifies the external behavior and internal implementation of each method executing a message and also describes the relationships between method specifications. For example, the message *sound-alarm* in a security system is implemented differently within different software objects**.** On receiving a *sound-alarm* message, for example:

- an air horn software object will engage the circuit to the air horn,
- a communications software object will send the name and address of the facility to the police, and

- a spotlight software object will turn on spotlights.

Design Rationale Component

The Design Rationale Component contains diagrams that describe milestone transformations of design artifacts throughout the design life cycle. The design rationale component allows the designer to record the reasoning for changes made to the design. Examples of the changes that may need to be documented in the design rationale component might include the following items:

- if the designer *re-partitions* the object class *person* to the object classes *person* and *employee* linked by a *subclass/superclass* relationship,

- if the designer *refines* the employs/employed by relation between the object classes employee and company by using referential attribute pointers.

Design Artifact Specifications

Each IDEF4 Design Artifact has a specification associated with it. The specification consists of external and internal specification components. The internal specification component is used for defining the internal implementation of the design artifact. The software engineer implements and maintains software with the internal specification. The external specification component is a «black box» or encapsulated view defining the external behavior, protocols, and responsibilities of the design artifact. The external specification is important to software engineers using and reusing the design artifacts or resultant software components because it defines the behavior of the design artifact without burdening the user with how the behavior is implemented. The external specification is also used for encapsulating legacy software and commercial «off the shelf» software components to make them usable in design specifications.

Summary

This chapter provided an introduction to the IDEF4 methodology and described (1) the IDEF4 object-oriented concepts, (2) the IDEF4 Design Method Procedure, and (3) the organization of a design in IDEF4.

The major focus of the chapter was on the use of object-oriented terminology as it applies to the IDEF4 method. Terminology introduced included: *object, method, message, class, subclass, instance, inheritance, encapsulation, abstraction, event, state,* and *polymorphism.* With an understanding of these terms, the IDEF4 method can be easily mastered.

The section on IDEF4 Procedure discussed the design process and how the activities involved in the design process are managed by the IDEF4 method and what the products of each stage will be.

The last section on IDEF4 Organization explained what models will be created in an IDEF4 design project and their purpose. An IDEF4 design project consists of three models: (1) Static Model, (2) Dynamic Model, (3) Behavior Model, and a Design Rationale Component.

IDEF4 MODEL ORGANIZATION

In IDEF4, design artifacts are grouped into three models: the Static Model (SM), the Dynamic Model (DM), and the Behavior Model (BM). The Static Model specifies the static structure of the design. The Dynamic Model specifies the dynamic communication processes between objects, classes, and systems. The Behavior Model specifies the implementation of messages by methods and contains a method diagram for each message detailing the relation between the method's external behavior and implementation. The Design Rationale is an added component that contains major transformations of design artifacts.

Each model represents a different cross section of the design. The three design models capture all the information represented in a design project, and the design rationale documents the reasoning behind the design. Each model is supported by a graphical syntax that highlights the design decisions that must be made and their impact on other perspectives of the design. To facilitate use, the graphical syntax is identical among the three models. For example, in all models, «®» is used to indicate that the artifact is derived or abstracted directly from the domain or real world, «Δ» is used to indicate that the artifact is in transition, and «©» is used to indicate that the artifact has reached final design specification. In all models, boxes indicate classes, round cornered boxes indicate instances, and arcs between boxes denote relations. No single model shows all the information encompassed in a complete design, and overlap among models ensures consistency. Each design artifact may also be associated with a formal specification of its external behavior and internal construction. The following sections introduce each of the models in more detail.

IDEF4 projects are organized into three design layers. Each design layer contains partitions. Each partition contains the three models and the design rationale component. The System layer represents the whole IDEF4 model. This is the actual product (i.e. the application. The Application layer contains the application specific partitions or solution specific objects. This may include actual code that has been generated and components that can be reused for other projects. The final layer is the Foundation Layer, which contains low-level objects. These could be buttons, forms, class libraries, and windows that constitute the software being designed.

At the System Layer, the software components are industry or domain specific (e.g. software components related to the banking industry). As the design moves to the Application Layer, the software components become specialized to objects supporting task specific designs. At the Foundation Layer, the objects being used are software mechanisms common across a number of industries (e.g. object libraries used by the medical industry, the banking industry, and the retail industry).

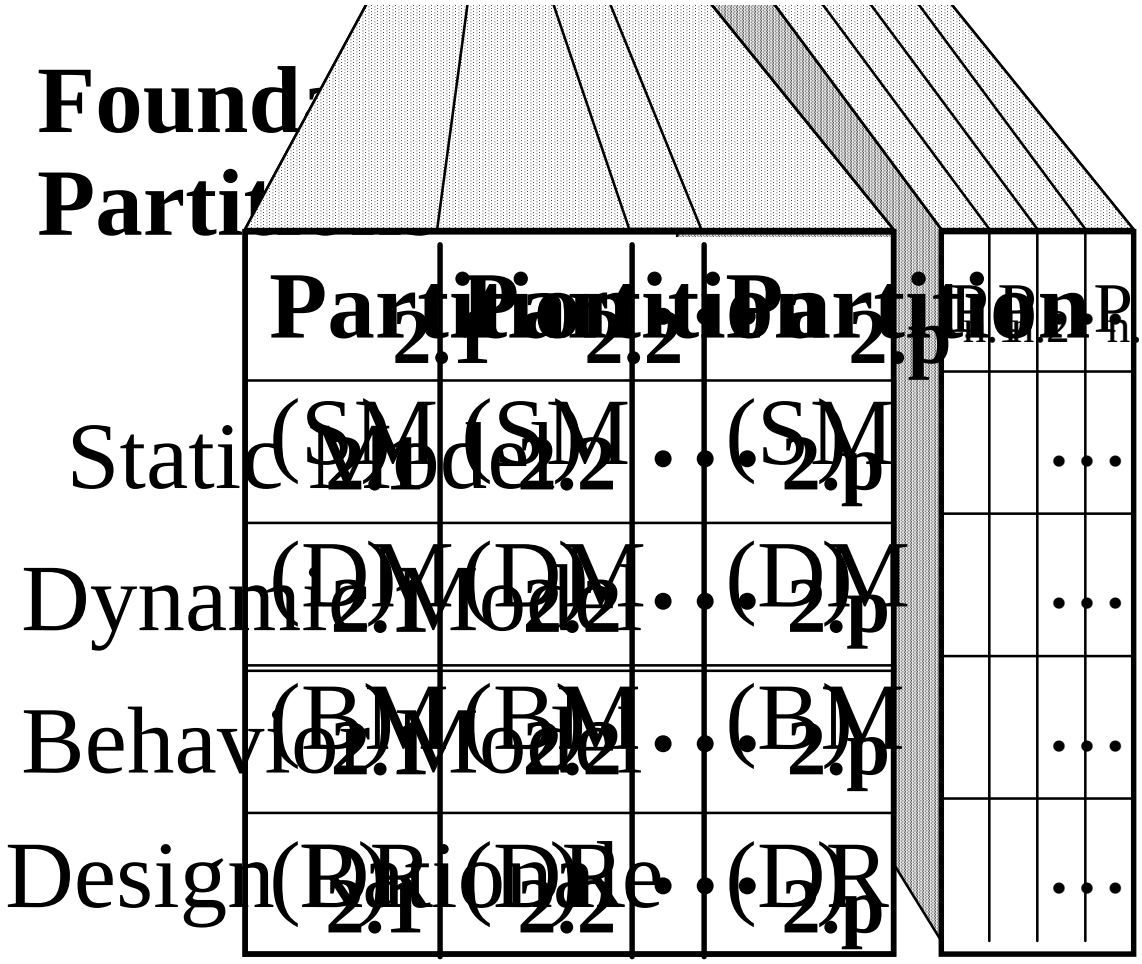


Figure 26
Organization of the IDEF4 Project

Each design partition must contain the three models and the design rationale component to document the different stages and to track where the design is at all times during the project. The models help transition the project through these three design partitions. Depending on the design situation, a project can have more or less than three design layers.

The Static Model

The first model created in any of the three design partitions is the Static Model. The Static Model specifies individual IDEF4 objects, object features, and the static object relationships. IDEF4 objects include instances, classes, and partitions.

The static model is made up of Structure Diagrams. The Structure Diagram depicts the relationships between objects. Structure Diagrams can be specialized to show specific types of relationships.

An inheritance diagram shows the class structure, specifying the inheritance relations among classes, class features, and information hiding. An Inheritance Diagram must contain at least one class. Figure 27 shows a class inheritance diagram that may be used by a company to distinguish between different people entering the facility.

IDEF4 Structure Diagrams

A structure diagram contains objects related by inheritance, links, and relationships. A structure diagram can be specialized to just include one type of relation structure. These diagrams would then be labeled by the relation structure type (i.e. inheritance diagrams, relation diagrams, link diagrams, and the instance link diagrams).

The specialized diagrams will be discussed in detail. The designer should keep in mind, however, that a diagram can be created that encompasses all of the elements.

IDEF4 Class Inheritance Diagram.

The triangles depicted on the relations point from the subclass to the superclass. In this diagram, the most general class is *Person*. The classes *Consultant*, *Employee*, and *Client* are specializations of *Person*. The classes *Hourly Employee* and *Salaried Employee* are specializations of *Employee*.

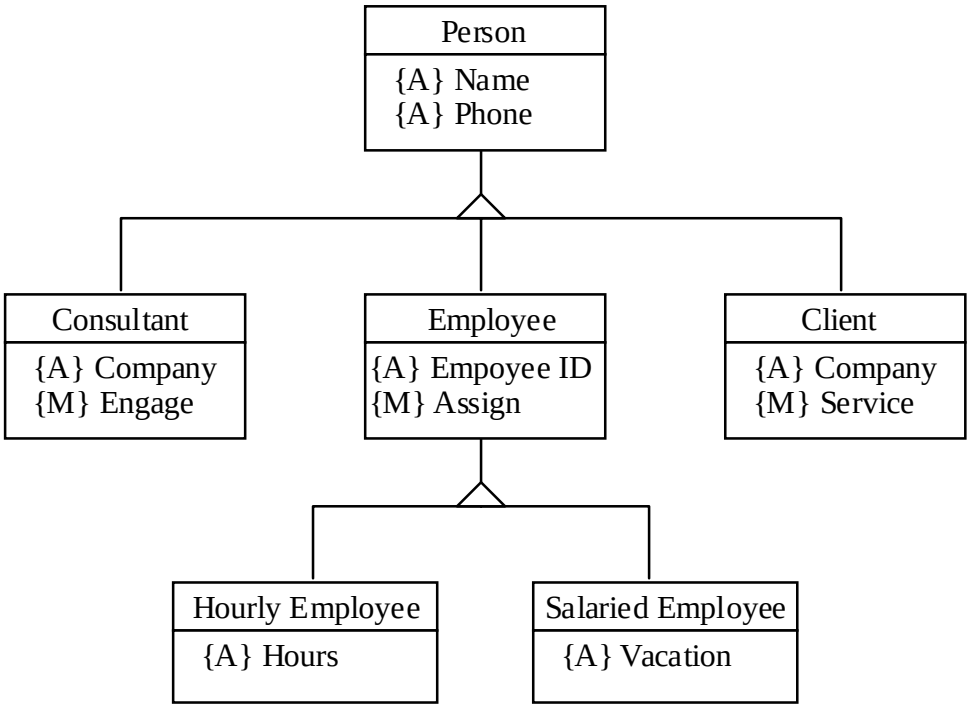


Figure 27
Inheritance Diagram

Information hiding is portrayed in an Inheritance diagram by sectioning the class box (Figure 28). The features listed below the first line in the class box are public; the features listed below the public features are private features.

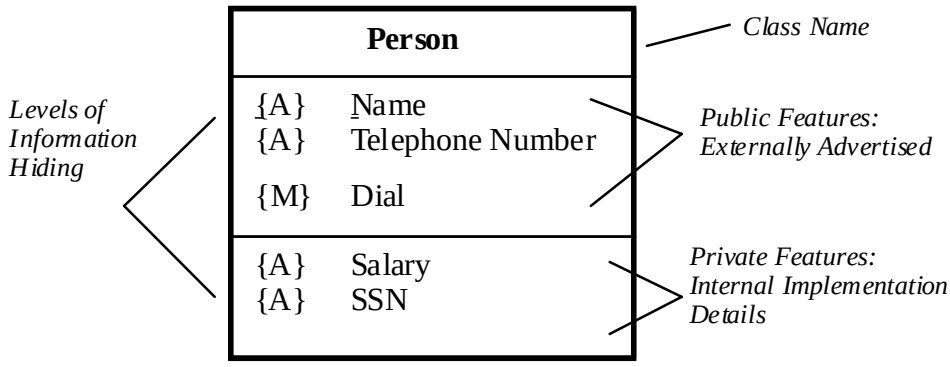


Figure 28
IDEF4 Class Box Showing Levels of Information Hiding

Figure 28 shows the class *Person*, which is being used in an automated telephone directory system. The person's name, telephone number, and the dial method are publicly accessible, but the salary and social security number are private. If no public/private bar is shown, then the features depicted are public by default. The public and private features of a class need not be shown if they do not add information to specific diagrams.

Relation Diagram

A relation structure is a simple relation between two objects. They can show, for example, a one-to-one or a one-to-many relationship. Figure 29 shows a relation diagram for the *Employed by/Employs* relation between *Person* and *Company*.

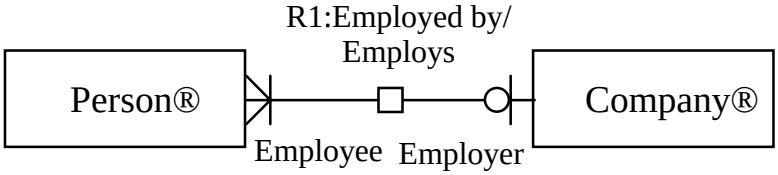


Figure 29
Relation Diagram

In this relationship, *Person* plays the role of *employee* and *Company* plays the role of *employer*. A company may have one or more employees, and a person may be employed by zero or 1 companies. This cardinality constraint is shown by the circle and fork (zero or more) on the Person side of the relationship and the circle and line (zero or one) notation on the Company side of the relationship.

Notice that the «r» noted after *Person* and *Company* represents an object that is from the application domain. Once this object becomes modified in the design evolution, the notation will change to «Δ.»

Link Diagram

As an object goes through design evolution, and the relationships should «disappear» because the designer has implemented the relationship. The implementation will be done using links, which are depicted in a Link Diagram. Figure 30 shows a Link Diagram containing a design evolution of the *Employed by/Employs* relationship to a link (L1). The link is achieved by imbedding the role names *employee* and *employer* as attributes in each class. The notation L1(R1) indicates that the link was derived from relation R1. Links in IDEF4 may be implemented by pointers, indices, or foreign keys. The imbedded attribute is known as a referential attribute because it refers to other object instances. Notice that the link notation is brought into the class box with a circle on the end to indicate that the relation has been embedded.

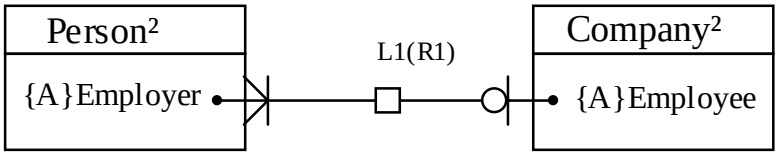


Figure 30
Link Diagram

The evolution from relation to link is tracked by the L1(R1) link label. There may be other constraints on the relationship R1 that will constrain methods that create, reference, update, and delete entities in the domain of the attributes defining the link. The IDEF4 method helps the project team keep track of design evolutions like this.

Instance Link Diagram

The validity of the Link Diagram is tested using the Instance Link Diagram. Object instances are run through the design describing real or possible object relationships. Figure 31 shows an instance link diagram with two instance of the class *Person*, [*John*] and [*Mary*] using the *Employed by/Employed* relation link *L1*. [John] and [Mary] work for the [ABC] company.

Figure 31
Instance Link Diagram

Object instances are represented by rounded boxes with a black dot in the center. Link instances are represented by connections with a black dot. The black dot serves as a visual cue for instances and is used in the IDEF3 Process Method and the IDEF5 Ontology Method. The Instance Link diagram is useful for challenging constraints in the design by exploring boundary conditions.

Behavior Model

The Behavior Model consists of method diagrams. Method diagrams show polymorphism and are used to take advantage of behavioral similarity by reusing method specifications.

Behavior Diagram

Figure 32 shows a method diagram highlighting the *Pay* method for employees. Permanent Employees are paid benefits with every paycheck and temporary employees receive pay with no benefits. This means that the method for calculating payment for permanent employees is different from the method for calculating payment for temporary Employees. The diagram highlights the difference between messages and the methods that implement the messages. For example, the message *Pay* is implemented by the method *BenefitPay* for permanent employees and by *StraightPay* for temporary employees.

Figure 32
Behavior Diagram

The Behavior diagram provides a view of a class of behavior across all the classes. This is useful for maintaining a consistent method signature (external protocols) or behavior across classes. The triangle on the relationship line denotes a generalization/specialization relationship which indicates that the behavior of *Perm_Employee.Pay* and *Temp_Employee.Pay* specializes the behavior and implementation of *Employee.Pay*.

Dynamic Model

The Dynamic Model has two kinds of diagrams, Client/Server Diagrams and State Diagrams. Both diagrams use asynchronous and synchronous communications to depict dynamic relations between objects. The dynamic relations are modeled using events and message passing.

Client/Server Diagrams

Client/Server diagrams illustrate the way objects use each other. IDEF4 uses Client/Server diagrams as a common syntax for describing the usage relationships between objects – from the system level of abstraction to low level objects. Figure 33 shows a client/server diagram depicting the dynamic payment relationship between *Calendar*, *Payroll*, *PermEmployee*, and *TempEmployee*.

Figure 33
Client/Server Diagram

The calendar generates events for month ends and week ends. Payment by *Payroll* is initiated by MonthEnd and End of Week events generated by *Calendar*. Note that the dashed arc indicates that the communication between *Calendar* and *Payroll* is asynchronous. The chevron on each link points in the direction that the event or message is being sent. The chevrons are shadowed to indicate that parameters are included. *Payroll* sends pay messages to permanent employees and temporary employees. The Pay message is synchronous as depicted by the solid line. The Pay message is implemented by the BenefitPay method in *PermEmployee* and by the StraightPay method in *TempEmployee*.

State Diagrams

The state model documents a set of states of the object and the state transitions. A state represents a situation or condition of the object during which certain constraints apply. The constraints may be physical laws, rules, policies, etc. The transitions are modeled using events and actions. Events are triggers which cause the object to initiate a transition from one state to another. The action is initiated on entry into a state (Figure 34).

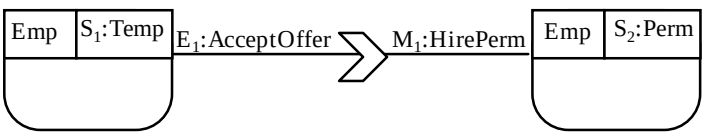


Figure 34

Employee State Diagram

The state diagram depicts the behavior of an object by showing the states an object can exist in and how it transitions from state to state. Using the example from the Client/Server diagram, an employee who is temporary may be hired on as a permanent employee. The TempEmployee object would have the method «Hire Permanent.» The «accept» event generated by a temporary employee is depicted on the relationship line. The message associated with this event, «Hire Permanent,» is depicted on the other end of the relationship line.

Design Rationale Component

An aspect of IDEF4 that sets it apart from other object-oriented design methods is its ability to explicitly distinguish between real-world objects, evolving design objects, and final design objects and its ability to record the evolution between these objects. The design rationale component of IDEF4 records major transitions in the evolution of an IDEF4 design. IDEF4 documents major design milestones as design states by recording the participating diagrams as a design state, and then giving the rationale for transition to another design state. The rationale is captured by describing the triggering observations and the resulting actions. Figure 35 shows three design states. Observations are normally symptoms or concerns about the design.

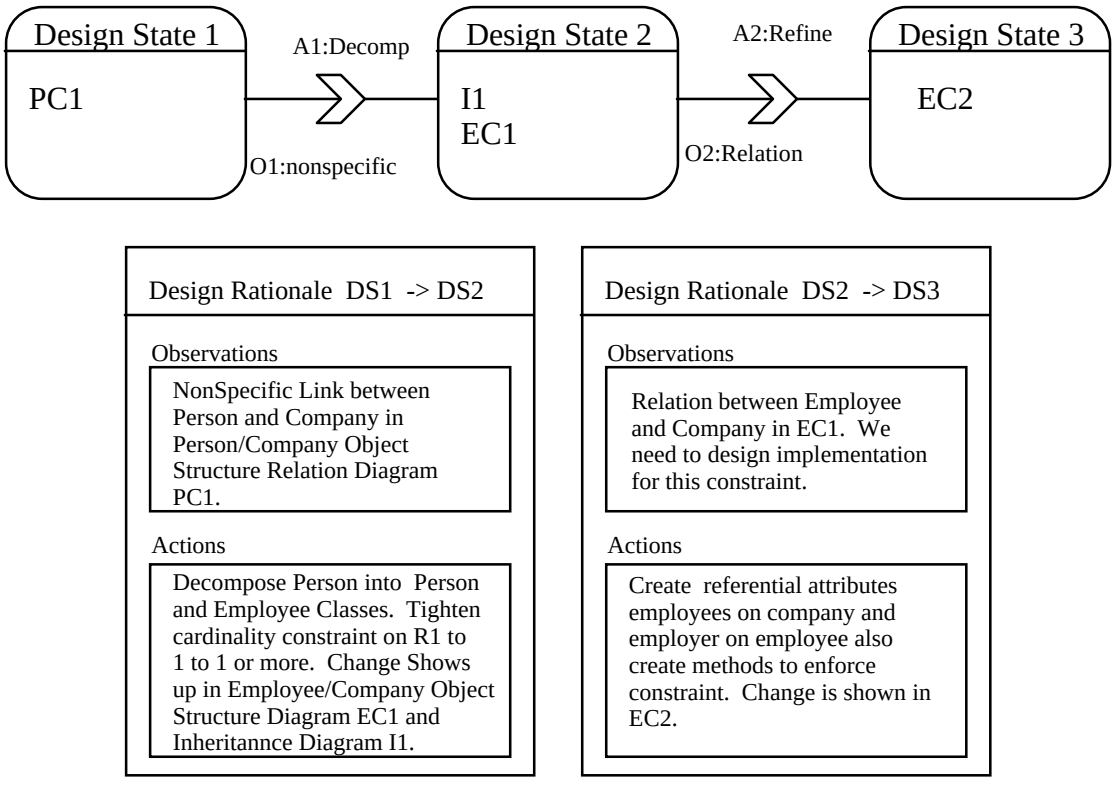


Figure 35
Design Rationale Diagram

Each design state lists the associated diagrams. The transition arc between design states has a set of observation events from a design state and a set of actions that is applied to go into the next design state. Thus rationale diagrams are actually specialized state diagrams.

Design Artifact Specification

Detailed specifications may be associated with any design artifact. There are two kinds of design specifications: the External Design Specifications (ExSpec) and the Internal Design Specifications (InSpec). The ExSpec defines the behavior of design features (Signature), whereas InSpec defines how the design object is to achieve that behavior.

A designer reusing a design object follows the external specification; a software engineer implementing the design follows the internal specification. IDEF4 promotes the view of software engineers as software component builders or software component users. The ExSpec enables the creation of component design libraries for design reuse.

Organization of an IDEF4 Project

IDEF4 Projects are organized in a hierarchical structure using IDEF4 Partitions. Each IDEF4 Partition is an independent model, having static, dynamic, and behavioral model components. This hierarchical structure of partitions and subpartitions is analogous to major sections, chapters, and subsections of a physical document and is suitable for the development of a document. Figure 36 shows an example of this hierarchical structure.

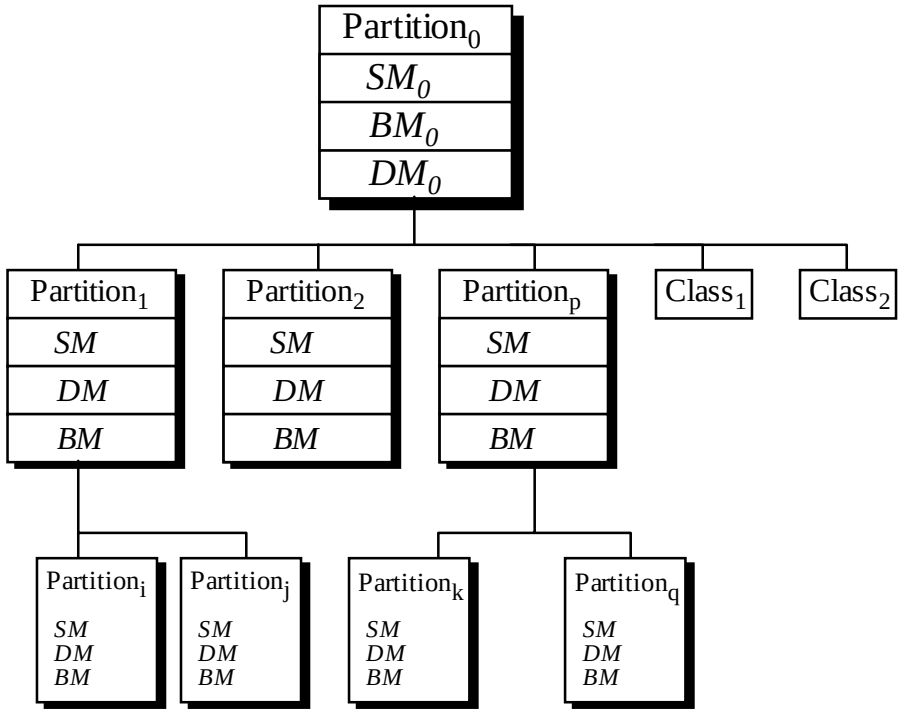


Figure 36 - IDEF4 Document Structure

Summary

IDEF4 has three models: (1) the Static Model (SM), (2) the Behavior Model (BM), and (3) the Dynamic Model (DM). A Design Rationale Component (DR) is included for recording the rational for major design evolutions. The SM specifies the static structure of the design, the DM specifies the dynamic communication processes between objects, classes, and systems, and the BM specifies the implementation of messages by methods and contains a method diagram for each message detailing the relation between method external behavior and implementation. Each design artifact may be associated with a design specification that specifies the external behavior (Signature) and internal implementation of the artifact.

STATIC MODEL

The Static Model (SM) specifies the static structure of the IDEF4 Object-Oriented Design. Each IDEF4 Design Partition must have a Static Model. The Static Model specifies individual IDEF4 objects, object features, and the static object relationships and links.

The Static Model contains structure diagrams. A structure diagram contains objects related by inheritance, links, and relationships. A structure diagram can be specialized to just include one type of relation structure, and would then be labeled by relation structure type, such as inheritance diagrams, relation diagrams, link diagrams and the instance link diagrams.

Creating the structure diagrams is the first step in an IDEF4 project. To create a structure diagram, object instances are identified and divided into classes. The relationships between these objects are identified. They may also be partitioned. Forms are then used to document information about the objects, classes, partitions, and relationships.

This chapter discusses object identification, IDEF4 Object naming conventions, examples of structure diagrams, and the forms used in the Static Model.

Object Class Identification

The IDEF4 Method assumes that the domain objects have been identified through Object-Oriented Domain Analysis. Methods such as IDEF1, IDEF5, IDEF3, SA/SD can be used to perform domain analysis (Yourdon, 1979). However, IDEF4 practitioners should be aware of how objects are identified, as the design process may reveal deficiencies in the Object-Oriented Analysis.

The identification of objects starts by focusing on the problem from the application domain and looking for the *things* in the problem. These things are likely to fall into five categories (Schlaer, 1988) that provide a useful place to start looking for objects:

(1) physical or tangible objects,

(2) roles or perspectives,

(3) events and incidents,

(4) interactions, and

(5) specifications and procedures.

Tangible or Physical Objects

These objects are often called *naive* objects because they are easy to find. For a given problem, it would be difficult to avoid recognizing automobile, taxi, airplane, train, dog, cat, landmark, radio, cellular phone, and book as objects. Figure 37 shows a set of physical objects.

Figure 37
Physical Objects

Objects Based on Role or Perspective

Objects take on different roles in different situations or when seen from different frames of reference. For example, it is possible for a person to take on many roles in different situations. These roles may be the person's permanent profession (e.g., lawyer, doctor, engineer, nurse, broker, accountant, or employee).

The role may be related to other activities that the person engages in (e.g., a patient in a hospital, a stockholder, a client, a trustee, a suspect in a burglary, or a tax payer). Frequently objects have many roles. For example, a doctor in a hospital could also be a patient in the hospital, a taxpayer, a licensed car driver, a credit card holder, a parent, and a pitcher on the local softball team. Figure 38 shows the different role objects passenger, nurse, and construction worker.

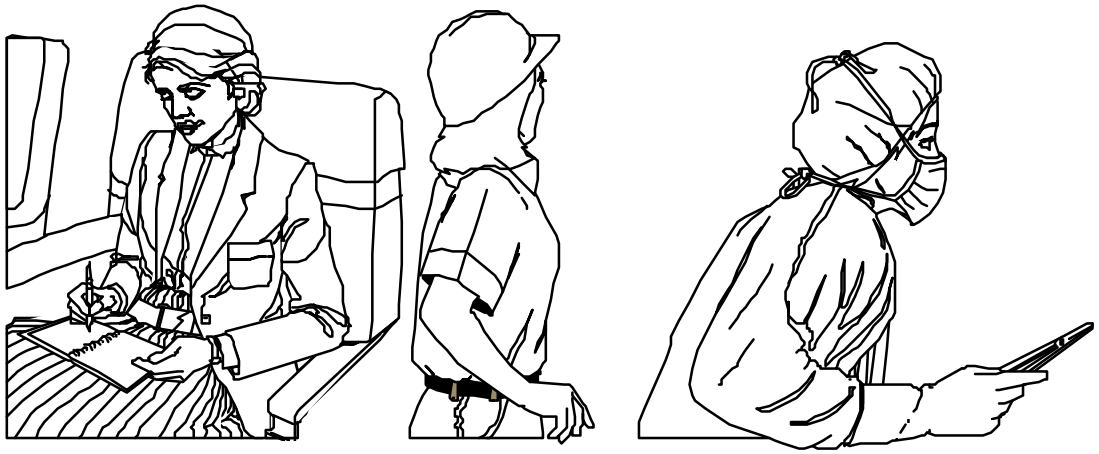


Figure 38
Role Objects

Events and Situations as Objects

Events or incidents may also be considered objects. The following are event objects: a horse race, a baseball game, a burglary, a car accident, a manufacturing defect, an earthquake, an election, a repair, and an airplane flight. The identification of events as objects is highly subjective, and will depend on the domain in which the software is to be used. Figure 39 shows events that could be considered as objects.

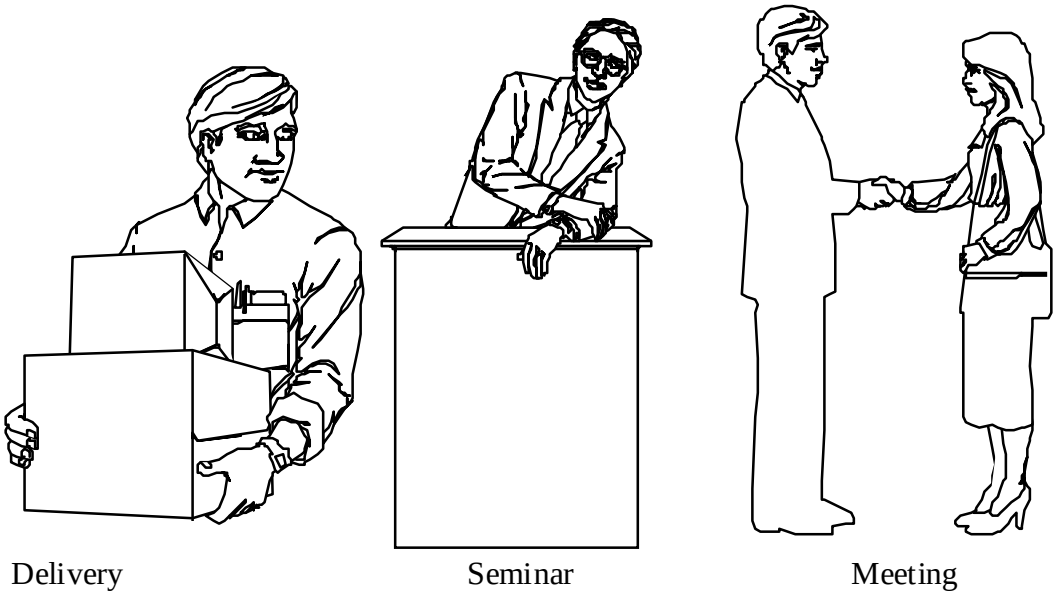


Figure 39
Event Objects

Interaction Objects

Interaction objects are the result of interactions or transactions between two or more objects. For example, the marriage certificate issued to a couple getting married and the sales receipt received by the customer from the sales person for an item purchased are transaction objects. Interaction objects can also arise out of modeling geometric objects (e.g., the intersection between two volumes, the edge between two faces, and the intersection point of two lines). Figure 40 shows a transaction object (i.e., the receipt).

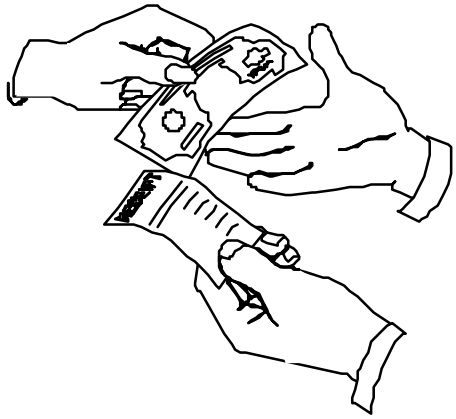


Figure 40
Interaction Objects

Interaction objects also arise in systems that involve flow (e.g., the piping in a water system, the lines in a telephone network, and the bus in a computer).

Specification and Procedure Objects

Specifications and procedures are found in all enterprises. ISO 9000 compliant manufacturers must have standard operating procedures (specifications for manufacture) for making products, inventory departments have procedures for replenishing inventory, products are built according to the specifications of the design department, and the sale of products are governed by standard operating procedures. Our lives are governed by laws, policies, and procedures. Specification objects describe the acceptable characteristics of objects instances. Procedure objects refer to the way other object instances may interact.

A drawing for a part (Figure 41) is a specification object for parts that are to be manufactured. A flow chart (Figure 41) is an example of a procedure object.

Figure 41
Specification and Procedure Objects

When searching for objects in a domain, the first objects that should be identified are physical objects. Role objects should be identified next, and so on, until specification and procedure objects have been identified. A domain may contain objects of all types or may just contain one type.

IDEF4 Object Naming Conventions

Object names that are clear and concise contribute to the clarity and explicitness of the models. It is preferable to name object classes with the common name used for object instances (e.g. «Person» used as a class for John, Mary, and Toonsie). However, the following problems may arise: (1) the name may be context sensitive (e.g., an *account* at a store and an account [story] of an experience), (2) the name may refer to more than one object class (e.g., *order* may refer to items being purchased by the company or items sold by the company), and (3) two or more names may be used for the same object class (e.g., *employee* and *worker*). In some cases, common names may not exist for certain object instances. In these cases, object class names must be coined.

Coining Terms

When coining terms, common terms for objects are preferred. In all other cases, names should be based on the essential nature of the object by concatenating common terms. For example, an official at a manufacturing equipment exposition could be coined as exposition-official or equipment-exposition-official. Often objects are named by a process or activity associated with them. This form of name coining should be avoided in IDEF4 models. An example of this is a restaurant which refers to customers by the food they order and their location. A phrase such as «The cheeseburger at four,» means the customer who ordered the

cheeseburger at table four. This type of naming convention[14] should not be used because it does not describe the object.

In coining terms, one should also examine the common terms of related object classes and ensure that the object class name «fits» with the related object classes. For example, if the terms bedroom, bathroom, and living room are in common use in a domain, it does not make sense to coin the term «storage environment» for a room that is used for storage purposes.

In IDEF4, the name of an object class can be used to uniquely identify it. For object instances the name of the class and a unique attribute of the instance must be used to identify it. Object instance names are constructed with the object class name followed by the value of the unique attribute in angle brackets. For example, *John*, an object instance of object class *Person*, is shown as *Person[John]*.

Objects that are scoped within partitions are referenced by *Partition ID'>'Object ID:*. For the partition class *Neural Net*, the *Node* class is referred to as *Neural_Net>Node.* The instance *Node[N23]* in *Neural_Net[NN2]* is referred to as *Neural_Net[NN2]>Node[N23]*.

Static Model Diagrams

Diagrams in the Static Model show different kinds of static relations between different kinds of objects. All of the relations can be depicted in one diagram, or specialized diagrams can be created for each relationship type. It is beneficial to be able to look at just the sub and superclass relations or just the inheritance, but it is often necessary to depict different kinds of relationships on a single diagram to give the designer an idea of how everything fits together. While a design is in transition, for example, it is necessary to depict relationships and links on a single diagram, in the event that the designer may not have decided on how to implement all of the relationships. A designer may also create a structure diagram that contains all the objects and relationships, and then break this out into more specialized diagrams.

To understand why both specialized and general diagrams should be developed, the developer should compare them to an assembly drawing. The actual assembly drawing gives a detailed road map of how everything fits together (a general diagram). The assembly drawing will also, however, have more detailed drawings of each section of the assembly which detail tolerances, measurements, and specifications (specialized drawing). Specialized diagrams, such as an Inheritance diagram, focus on one aspect of the design relationships. General diagrams let the designer view these relationships in one place so that when the design is in flux there will be just one diagram changing, as opposed to having several diagrams would then need to be updated.

[14]Known as metonymy.

Structure Diagrams

DEF4 uses Structure Diagrams to define the structural relations between objects. Besides the Inheritance diagram, there are three other kinds of Structure Diagrams, categorized by the kind of object and kind of relation that is used: (1) Relation Diagrams (Figure 43), (2) Link Diagrams (Figure 44), and (3) Instance Link Diagrams. The practitioner need not be aware of these kinds of structure diagrams, as the type of diagram is directly implied by the kind of objects and kind of relations used in the diagram. Figure 42 shows the kinds of IDEF4 relationships.

Figure 42
Kinds of IDEF4 Relationships

A relation consists of a line connecting two classes or two object instances with a smaller box containing a unique identifier for the relationship. A rolename may be assigned to either end of the link to indicate the way that the class participates in the relationship. If the relation box is shadowed, it indicates that other auxiliary objects participate in the relationship. For example, the relationship between the owner of a house and the house involves a title deed, which would be an auxiliary object.

Each relation and link in a structure diagram has multiplicity or cardinality constraints on both ends (Figure 42). Each constraint specifies the maximum and minimum cardinality of the relation or link. For example, Figure 43 may be read as every instance of object1 is associated with one and only one instance of object2, and every instance of object2 is associated with zero or more instances of object1.

Figure 43
Structure Diagram Relation/Link Symbols

Figure 44 enumerates all combinations of cardinality constraints and shows examples of each.

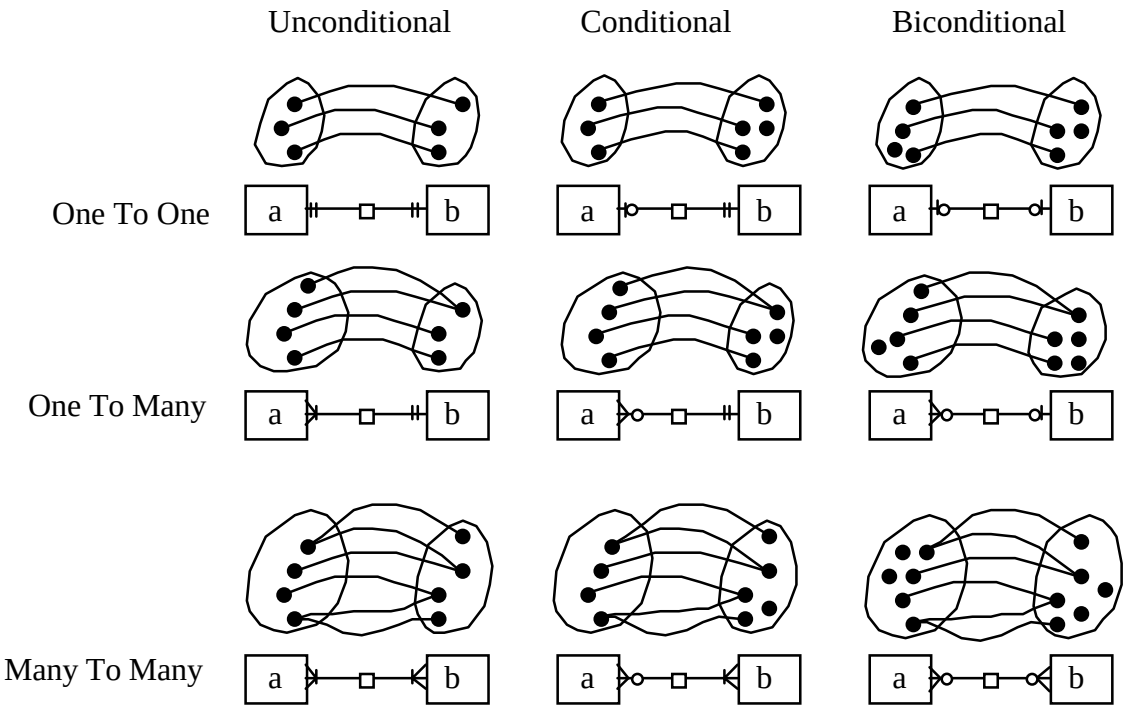


Figure 44
Structure Diagram Multiplicity Constraints

There are three forms of cardinality constraint: one-to-one, one-to-many, and many-to-many. A one-to-one relationship exists if an instance of a class is associated with a single instance of another. For example, in the class *Social Security Card* one instance of Social Security Card would be associated with an instance from the class *Person*. A one-to-many relationship exists if an instance of a class is associated with one or more instances of another, and if each instance of the second class is associated with just one instance of the first. An example of this would be the class *Computer* in which many instances of the class *Person* shared

the same computer. A many-to-many relationship exists when an instances of the first class are associated with many instances of the second class and vice versa. This might occur if the class *Employee* had many employee instances that could work on an instance of a project from the class *Project*, and an instance of a project could have many employees working on it.

When all of the instances in two classes participate in associations, the relationships are called unconditional. If some of the instances from one of the classes in a relationship do not participate, then the relationship is called conditional. If some of the instances from both of the classes in the relationship do not participate in the relationship, then it is called biconditional.

Cardinality is important to the designer because the implementation used for the system should be decided by the relationship's cardinality. For example, a one-to-one relationship would not be implemented the same way a many-to-many relationship is implemented. However, for each type of cardinality there are many different ways for a relationship to be implemented.

Once the designer has a grasp of the concepts of relationships, rolenames, and cardinality, a structure diagram can be built. The next sections discuss the specialized diagrams. The Inheritance Diagram shows inheritance relations between class and partition objects. The Structure Relation and the Link Diagrams show static relations and links between classes and between instances.

Inheritance Diagrams

Inheritance diagrams show subclass/superclass relations among classes. The concept of inheritance provides a means of organizing object instances into related sets (classes), which allows for the reuse of methods and class features.

From the real-world perspective, the inheritance phenomenon operates like a specialization relation. That is, the inheriting class (subclass) is a specialization of the class from which it inherits (superclass). Figure 45 shows the IDEF4 inheritance relations.

Figure 45
Kinds of IDEF4 Inheritance Relations

The open triangle indicates that subclasses are mutually exclusive (disjointed). An example of this would be the subclasses *Star* and *Planet* for the superclass *Celestial Object*. Because of the different physical and chemical make up of these two subclasses, a planet will never be a star and a star will never be a planet so these classifications are disjointed.

The filled triangle indicates that subclasses overlap. For example, *car* and *boat* are subclasses of *vehicle*. An amphibious car is an instance of car and boat so the subclasses are overlapping and not mutually exclusive because an instance of an object can belong to more than one class.

IDEF4 also allows specialized notation to indicate whether an internal specification (specification of internal implementation, i.e., code) or signature (implementation of external behavior, i.e., a protocol) is being inherited. An «i» near the inheritance triangle indicates inheritance of internal specification, whereas an «e» near the triangle indicates inheritance of signature.

Figure 46 shows an inheritance diagram for different classes of oven. The Oven Class has methods defined for switching on the oven, switching off the oven, and determining the temperature of the oven. The '!' in front of the «Switch On» and «Switch Off» methods indicate that the method is redefined in the class. The Microwave Subclass specializes the behavior of the Oven, by being able to set cooking time, and by having different methods for switching on and off. These two methods are said to «shadow» the methods in Oven because they achieve the same end result (turning the oven on and off) but it is done by a different means (i.e., method).

Figure 46
Inheritance Diagram for Ovens

The *Electric Oven* specializes *Oven*, but the methods for switching it on and off are the same as for oven with some additional constraints. This is indicated by the '+' in front of the Switch On and Switch Off methods in *Electric Oven*.

The *EasyMicrowave* has a new specification for SetTime indicated by the '!' prefix, but uses the same Switch On and Switch Off methods used by *MicrowaveOven*, its superclass. *ConvectionMicrowave* inherits behavior from *Electric Oven* as well as *MicrowaveOven*. Open Door and Close Door are redefined in *ConvectionMicrowave*, but it is not clear whether Switch On and Switch Off are inherited from *MicrowaveOven* or *ElectricOven*. Name conflicts like these must be resolved in the MicrowaveOven Class Specification.

Relation Diagram

The Relation Diagram shows relations between objects. Each relation has a relation label, a unique identifier, a role label from each object attached to the relation, and the cardinality of each object attached to the label. Each relation contains a small box to distinguish it from a link. Figure 47 depicts the relationships R1, R2, and R3 between the objects desk lamp, base, lamp shade, and light bulb. A desk lamp has one base (R3), one lampshade (R2), and one or more light bulbs (R1). A base, light bulb, and lamp shade are associated with only one Desk Lamp.

Figure 47
Relation Diagram

Link Diagram

A link is an implementation of a relation using referential attributes. The Link Diagram shows the links between referential attributes. A link points from a referential attribute on an object to the domain object of the attribute. If the link is bi-directional, then it links the attributes on both attributes' participating objects. Each link has a unique link identifier, an optional reference to a relation, and the cardinality of the participating objects. The optional relation reference is used to indicate the relation from which the link was derived. Figure 48 shows a link diagram derived from the relation diagram shown in Figure 47. The Object Class DeskLamp has referential attributes *base*, *shade*, and *bulbs* that point to instances of *base*, *lamp shade*, and *light bulb*.

The relation reference is shown in parenthesis after the link identifier. The notation L1(R1) indicates that the link was derived from relation R1. Links in IDEF4 may be implemented by programmers as pointers, indices, or foreign keys. The attribute is known as a referential attribute because it refers to other objects.

Figure 48
Link Diagram

Notice that the link notation is brought into the class box to indicate that the relation has been embedded. The evolution from relation to link is tracked by the L1(R1) notation because there may be other constraints on the relationship R1 that will constrain methods that create, reference, update, and delete entities in the domain of the attributes defining the link. The IDEF4 method helps the project team keep track of design evolutions like this.

Instance Link Diagram

It is often useful to validate the design by describing real or possible object relationships**.** An Instance Link Diagram shows scenarios using actual instances of a class to validate relationships. The designer can verify that the links created in a Link Diagram are correct using instances or actual occurrences of an object.

Figure 49 shows an instance link diagram for the link diagram shown in Figure 48. *DeskLamp 23* points to *Base 21*, *LampShade 33*, and *LightBulb 21* through referential attributes *base*, *shade*, and *bulbs* respectively.

The Instance Link diagram is useful for challenging constraints in the design by exploring boundary conditions.

Figure 49
Instance Link Structure Diagram

Object Structure Specifications

After creating structure diagrams, the designer will have a good idea of which objects are valid objects in the design. For each object identified in the design, IDEF4 requires an external and internal class specification. The specification contains the object name, a description of the object, the rationale for abstraction, and a list of its superclasses (Table 1). The external portion of the object specification contains descriptions of the object's public features. The internal specification contains specifications for private features.

Table 1. Object Specification Form

Author	*J. Smithe*	**Project**	*IICE*	**Date**	*6/1/94*	**Revision**	*3*
Object Specification		**Name**	*Object_Name*	**Partition(s)**		*Partition Names*	
Description	*A brief description of the object*						
Rationale	*Rationale for abstracting the object*						
Super Classes	*A list of the object's superclasses*						
Diagrams	*A list of the diagrams in which the object is used*						
External Features							
Partition/Class/Instance		**Event**		**Message**			
A list of embedded objects		*A list of events*		*A list of messages*			

Attribute	**Description**	**Role**	**Domain**
Attribute Name	Short description of attribute	Is the attribute referential, descriptive, or identifying	The attributes' domain
Internal Features			
Partition/Class/Instance	**Event**	**Message:Method**	
A list of embedded objects	*A list of events*	*A list of messages*	
Attribute	**Description**	**Role**	**Domain**
List of Attribute Names	*List of corresponding attribute descriptions*	*Is the attribute referential, descriptive, or identifying*	*List of corresponding attribute domains*

Table 2 shows an object specification form for the employee. An employee is a person who works for a company. The employee class appears on inheritance diagram ID21, structure diagram SD23, and link diagram LD22. Employees can generate events for strikes and injuries. Employees can receive messages for work and travel that have been implemented by the methods work and travel respectively. Employees have a name, employee identity number, and department. Employees have a salary, but that is hidden from the general public.

Table 2. Object Specification Form Example

Author	*J. Smithe*	**Project**	*IICE*	**Date**	*6/1/94*	**Revision**	*3*
Object Specification		**Name**	*Employee*	**Partition(s)**		*Company*	
Description		*An employee is a person that works for a company*					
Rationale		*Employees are 'physical objects' that appear in the company partition*					
Super Classes		*Person*					
Diagrams		ID21, SD23, LD22					
External Features							
Partition/Class/Instance		**Event**		**Message**			
none		*strike, injury*		*work: work* *travel: travel*			

Attribute	Description	Role	Domain
Name	The name of the person	descriptive	String
ID	Unique employee ID	Identifier	Integer
Dept	Department that employee works in	Referential	Department
Internal Features			
Partition/Class/Instance	**Event**	**Message**	
None	*None*	*reminder*	
Attribute	**Description**	**Role**	**Domain**
Salary	*The monthly Salary of the employees*	*Descriptive*	*Currency*

The IDEF4 method also requires other elements of the diagrams to be documented as well. If the object shown in Table 2 is a partition, then a Partition Specification Form (Table3) must also be filled out. The Partition Specification Form includes information on the Static Models, Dynamic Models, and Behavior Models of the partition. The Partition Specification also contains lists of specifications for each of the three models.

Table 3. Partition Specification Form

Author	*J. Smithe*	**Project**	*IICE*	**Date**	*6/1/94*	**Revision**	*3*
Partition Specification		**Name**	*Company*	**Partition(s)**		*Conglomerate*	
Physical Allocation		**Node**	*N241*	**OS**		*UNIX™*	
Static Model		**Dynamic Model**		**Behavior Model**			
LD22 SD23, ID23		*CS21, CS19, SM10, SM11*		*BM20, BM21*			
EmployeeSpec, DepartmentSpec		*EmployeeHiringSpec*		*workspec travelspec*			

For each relationship specified in the design, IDEF4 requires a form containing the name, description, roles, and cardinality of the relation. Table 4 shows the form for specifying relations. If the partitions are tagged with network node designators, then the physical partitions of the generated software are specified.

Table 4. Relationship Specification Table

Author	*J. Smithe*	**Project**	*IICE*	**Date**	*6/1/94*	**Revision**	*3*
Relation Specification		**Name**	*Relation Name*			**ID**	*UniqueID*
Description	*Description of relation*						
Implementation	*List of diagrams and specifications detailing relation implementation*						
Object/Relation	**Role**	**Relation**	**Cardinality**				
Object Name	*Role Name*	*Relation Name*	*Cardinality Constraint*				
Object Name	*Role Name*	*Relation Name*	*Cardinality Constraint*				
Object Name	*Role Name*	*Relation Name*	*Cardinality Constraint*				

For each link specified in the design, IDEF4 requires a form containing the Link ID, description, roles, cardinality of the link, and the relation from which the link was derived. Links can be derived from relations by creating referential attributes from the relation's role names.

The Works_for/Manages relation form shown in Table 5 shows the relationship between employees and managers. The relationship is shown implemented as a link in link diagram LD24 and link specification in LS21. An employee acting as a worker works for a manager and a manager acting as a supervisor manages one or more employees.

Table 6 shows the specification of the implementation of this relationship using a link. The works for part of the works_for/manages relation has been implemented using a pointer from employee to manager and using the manager referential attribute of employee.

Table 5. Relationship Specification Table for Works_for/Manages

Author	*J. Smithe*	**Project**	*IICE*	**Date**	*6/1/94*	**Revision**	*3*
Relation Specification		**Name**	*Works_For/Manages*			**ID**	*RS21*
Description	*An employee works for a manager a manager manages 1 or more employees*						
Implementation	*LD24, LS21*						
Object/Relation	**Role**	**Relation**	**Cardinality**				
Employee	*worker*	*works for*	*one and only one manager*				
Manager	*Supervisor*	*manages*	*one or more employees*				

Table 6. Link Specification Table Example

Author	*J. Smithe*	**Project**	*IICE*	**Date**	*6/1/94*	**Revision**	*3*
Link Specification		**Link Name**	*Works For*			**ID**	*LS21*
Description	*A pointer from employees to supervising manager*						
Relation	RS21:Works_for/manages						
Object		**Attribute**		**Cardinality**			
Employee		*Manager*		*1 manager*			
Object Name		*Referential Attribute Name*		*Cardinality Constraint*			

Advanced Features

This section discusses advanced features which relate to the different scenarios that occur when using inheritance. These specializations are specialization based on (1) external protocol, (2) internal implementation, and (3) sub- and super- type.

Advanced Features of Inheritance

Normally, when one class inherits features from another class, the external features and internal features are inherited. In some situations, only the external signature of the class is inherited or inheritance of external and internal features occur in opposite directions. IDEF4 allows additional notation on the inheritance relation to model these situations.

Specialization Based On External Protocols

Under this specialization, the subtype inherits its external specification from the superclass (i.e., the subclass specializes the external protocol of the superclass through inheritance). The inheritance diagram shown in Figure 50 shows that the classes *Fluorescent Bulb* and *Incandescent Bulb* specialize the external signature of *Light Bulb*. The inheritance symbol is labeled with an 'e' to denote external inheritance.

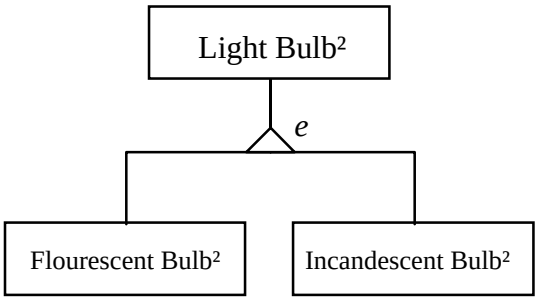


Figure 50
Subtype/Supertype External Inheritance Relationship

Specialization Based on Internal Implementation

The subclass inherits internal specification from the superclass (i.e., the subclass uses the implementation code of the superclass through inheritance). The inheritance diagram shown in Figure 51 shows that the classes *Fluorescent Bulb* and *Incandescent Bulb* inherit external signature. The inheritance diagram shown in Figure 51 shows that the classes *Fluorescent Bulb* and *Incandescent Bulb* inherit and specialize the internal implementation of light bulb as well as the external signature. The inheritance symbol is labeled with an 'i' to denote internal inheritance

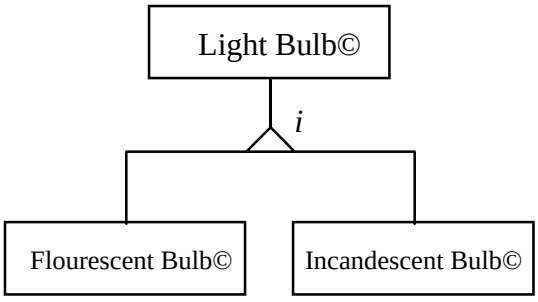


Figure 51
Subclass/Superclass Internal Inheritance Relationship

Specialization Based on Subclass and Sub Type

Normally inheritance of internal and external signature are in consonance; i.e., the subclass specializes the internal implementation of the superclass, and the subclass specializes

the external signature of the superclass. This is default interpretation of the triangle inheritance symbol (Figure 52).

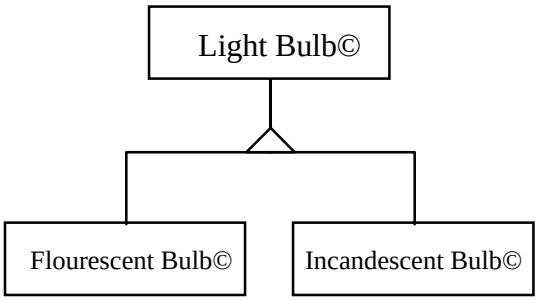


Figure 52
Subclass/Superclass Inheritance Relationship

In rare cases, the external and internal specialization relations do not correspond: Real Numbers are a subset (external specialization) of Complex numbers, yet in a design, for practical implementation considerations, the internal specialization relation may be the reverse of subtype/supertype. Complex numbers have a real and imaginary component, but real numbers do not have an imaginary component (its value is zero). In terms of internal implementation, we would model complex numbers as a subclass of real so that the complex number class could inherit the real attribute from the real number class and specialize it by adding an attribute for the imaginary component. However, from an external point of view, we would like all of the arithmetic operations on real and complex numbers to work correctly so we would model real number as a specialization of complex number when looking at the external signature. The Inheritance diagram in Figure 53 models this situation. The triangle labeled ‘e’ and the triangle labeled ‘i’ point to the superclass seen from an external and internal point of view, respectively.

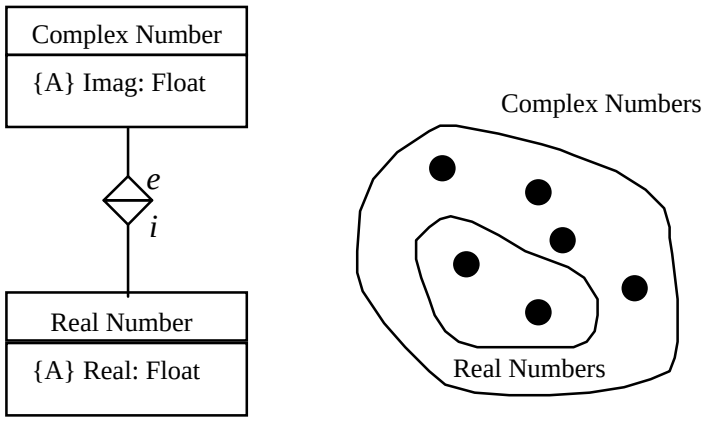


Figure 53
Inheritance Relationship of Complex and Real Numbers

Advanced Features of Relations

Normally relationships are binary; i.e., they involve two objects. Relationships, however, are not always binary: IDEF4 can describe relationships that involve more than two objects, relationships that involve objects, and relationships that involve relationships. An example of these relationships are: (1) the married relationship between husband and wife that involves a marriage certificate; (2) the house owner relationship involving person, house, and title; and (3) the assignment relationship between person, project, and skill.

Figure 54 depicts the ternary relationship between person, dog, and license. This relationship involves the dog license object. A person may own dogs and a dog may be owned by a person. To own a dog, a person must have a dog license. The relationship box is shown shadowed to indicate that a relationship object is involved. N-ary relationships (relationships involving more than two objects) do not occur frequently, but if they do, then they should be resolved in terms of binary relationships, or relation objects, or the relationship should be «objectified.» Cardinality cannot be specified on n-ary relationships.

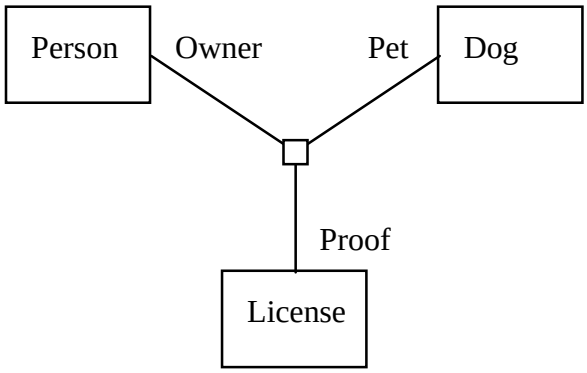


Figure 54
Ternary Relation Involving Person, Dog, and License

Figure 55 shows this relationship modeled as a binary relationship between Person and Dog with a secondary relationship between License and the Owns/Owned By relationship. A person may own a dog, and a dog may be owned by a person, and every Owned/Owned By relationship may have one license. A license must be associated with an Owns/Owned By relationship. The secondary relationship is often used to resolve many-to-many relationships.

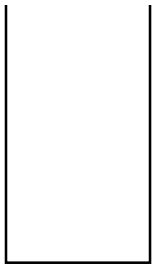


Figure 55
Secondary Relationships

The shadowed relationship box shown in Figure 56 denotes relationships that involve objects. This relationship box may be used as an abbreviation in diagrams for the more explicit form shown in Figure 55.

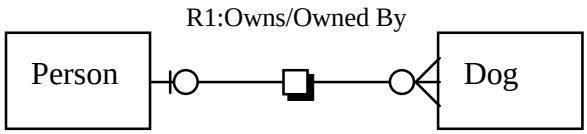


Figure 56
Relation Involving Objects

Figure 57 shows a possible link diagram for the Person/Dog/License Relation. The link diagram contains redundant links which may be used to increase efficiency.

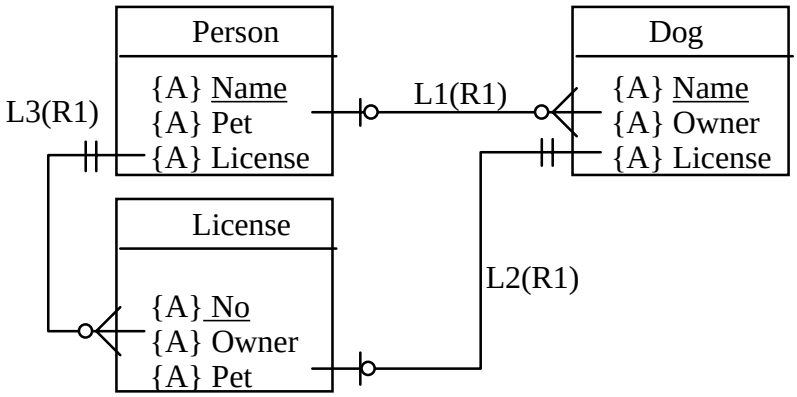


Figure 57
Link Diagram For Person/Dog/License Relation

Class Features

A class is a special kind of object that is known as a *class object* or *metaobject*. Sometimes it is necessary to define features on the class object. These features are known as *class features*. A variable that counts the number of created instances of a class is a good

candidate for a class attribute. Methods that create instances are really class methods because they belong to and operate on the class object, and because the class object has the information necessary to instantiate objects. Class objects can also have links to its subclass and superclass objects, but these links are typically private.

In IDEF4, class features have a box drawn around them to indicate that they do not apply to instances of the class, but to the class object itself. Class attributes are useful for storing information that is the same for all instances; for example, the attribute *sides* is always three for instances of triangle and the number of instances that have been created from a class should also be common for all instances of the class. The number of instances of an object that have been created by a class object is information that can be shared across all instances. Figure 58 shows a triangle class with conventional *area* and *vertex* attributes. Triangle also has class attributes *sides* for the number of sides of a triangle and *instances* which count the number of instances created. All instances of triangle have access to this attribute.

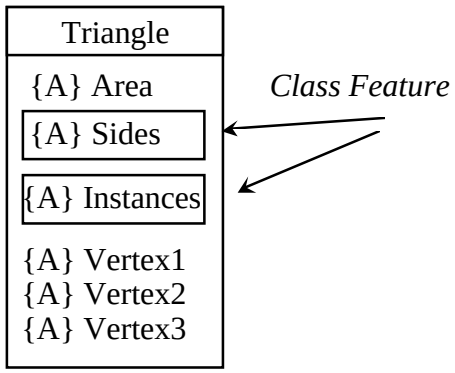


Figure 58
Representation for Class Features

Alternate IDEF1X Cardinality Syntax

The choice of the «crows feet» cardinality notation in IDEF4 was intentionally designed to appeal to the large commercial base of entity/relationship (ER) modelers who are familiar with this notation. There is also a large base of modelers familiar with the «black dot» cardinality notation of the IDEF1X semantic data modeling method. In recognition of this large base of IDEF1X users, IDEF4 allows IDEF1X's «black dot» cardinality notation to be substituted for the «crows foot» notation. The two cardinality notations are illustrated in Figure 59 using the Owns/Owned By relation between instances of the class *Person* and instances of the class *Dog*. Both examples express the same relation: a person owns zero or more dogs and a dog is owned by one and only one person.

Figure 59
Crows Feet and IDEF1X Syntax

There is a one to one mapping between the cardinality symbol sets, so anything that can be expressed using the ER cardinality notation on the relations in an OOD can be expressed using the IDEF1X cardinality notation on the relations in an OOD, and vice versa.

Users of the IDEF1X cardinality notation in IDEF4 should be careful of confusing the semantics of object-oriented modeling with semantic data modeling. The IDEF1X concepts of entity, primary key, foreign key, key attribute, and parent/child relationship do not map directly to OOD. For example, objects do not need identifiers (primary key). Objects do not have foreign keys. In IDEF4, the equivalent object concept for parent and child entity is not explicitly supported, but may be derived from the cardinality constraints.

Figure 60 enumerates possible combinations of the IDEF1X cardinality symbol for one-to-one, one-to-many, and many-to-many relations against unconditional, conditional, and biconditional cases. Figure 44 shows the «crows feet» notation equivalents for the «black dot» notation shown in Figure 59.

Figure 60
IDEF1X Cardinality Symbols used in IDEF4

Use of the IDEF1X «black dot» notation will make it easier for IDEF1X modelers to transition their models into OOD. IDEF1X models may be cast as object-oriented models in IDEF4 by re-interpreting the semantics of entity to mean Class, relation to mean link, categorization to mean inheritance, foreign key to mean referential attribute, and key attribute to mean naming attribute. The ability to migrate from IDEF1X to IDEF4 could be useful for organizations that have large inventories of IDEF1X models and would like to leverage this investment in object-oriented design.

The IDEF1X method was developed for designing relational database schemas; therefore, IDEF1X models exhibit a commitment to a certain style of implementation. Thus it may be necessary to reverse engineer IDEF1X models to translate to an IDEF4 object model. Entities in IDEF1X are identified on the basis of descriptive attributes (i.e., state), whereas in IDEF4, objects are identified on the basis of their behavior, leaving the attributes as an implementation decision. IDEF1X associative entities are identified to resolve many to many relations; therefore, from an OOD perspective, they may appear contrived. IDEF1X relations represent an implementation choice, that is why IDEF1X relations are equivalent to IDEF4 links. IDEF4 links are an implementation decision based on IDEF4 relations, thus IDEF1x relations must be reverse engineered to find the IDEF4 relation underlying the link. Figure 61 illustrates this point by showing the IDEF1X *Employs/Employed By* relation with its equivalent IDEF4 link. Note the referential attribute plays the same role as the foreign key in the Employee class. Because the IDEF1X relation is one of many possible implementations, it is important to define the IDEF4 relation on which the IDEF4 link could be based. The IDEF4 relation is important in that it will allow a common take-off point for investigating other implementation options.

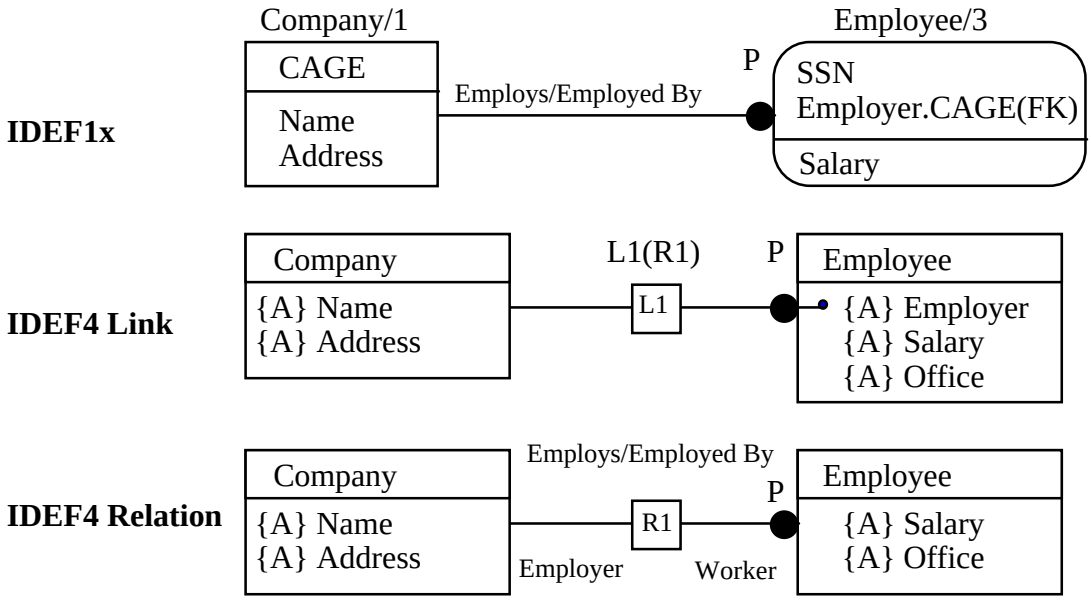


Figure 61
IDEF4 Equivalent Representations to IDEF1X Relation

In IDEF1X, the foreign key is imbedded in the dependent entity because relational databases rely on queries for joining data across tables. OOD queries are performed by navigating across the network of links between objects, so that the placement of links using referential attributes (attributes that refer to other objects) is strongly influenced by the intended usage of the objects and are not necessarily imbedded in the dependent object as they would be in IDEF1X.

The categorization relationships in IDEF1X can give the designer an idea of possible subclass/superclass relationships. The categorization relationship also depicts which attributes can be stored by the superclass and do not need to be included in the subclass. Figure 62 shows the equivalent IDEF4 syntax for an IDEF1X categorization.

Figure 62
IDEF1X Categorization and Equivalent IDEF4 Inheritance

In summary, use of the IDEF1X «black dot» cardinality symbol should lower the semantic impediment to object-oriented design by experienced IDEF1X modelers, and should also serve to encourage the use of the existing inventory of IDEF1X semantic data models.

BEHAVIOR MODEL

The Behavior model classifies methods by behavioral similarity. A behavior diagram classifies a specific system behavior type according to the constraints placed on the methods implementing a message. The arrows indicate additional constraints placed on the method sets. The Behavior model consists of behavior diagrams. Each behavior diagram depicts instances of a kind of behavior.

Strictly speaking, a method in an OOPL is described by its code; in IDEF4, a method is described in a behavior specification which is like a contract for its implementation. The behavior specification contains an external and internal component. The external behavior specification is a declarative statement of the intended effect of the method. For a non-side-effecting function, the specification would state the relationship between the function argument list and the corresponding return values. For a procedure or a side-effecting function, the specification would also define how the method changed the entire state of the world when given an argument list and a prior state of the world. The method contract may in some situations contain algorithmic restrictions such as, «The move method will execute an erase method followed by a draw method.» A specification might also give other information about the intended nature of the method (e.g., its time complexity). In addition, one would expect to find statements of actions that should occur before a method is invoked and what actions should occur after the method has performed its task. A behavior specification provides the constraints that are specified to hold for all implementations of the method.

Behavior Diagram

The purpose of the Behavior Diagram is to describe relationships between internal and external specifications of behavior patterns (families of behavior). The behavior family can be either globally scoped or scoped within a partition.

The intent of behavior diagrams is to provide a separation of the method hierarchy from the class hierarchy, to reuse method hierarchy and specifications for other classes and designs, and to focus on behavior independence of object model (only at the end do we link methods to classes).

Behavior Diagram Syntax

The behavior diagram consists of round-cornered boxes denoting behaviors and links between these boxes that in turn denote generalization/specialization relations between behaviors. The message name is entered at the top section of the box, the participating classes are listed in the center section, and the method signature is specified at the bottom. The signature of the method includes the method name, input parameter types, and return types (Figure 63).

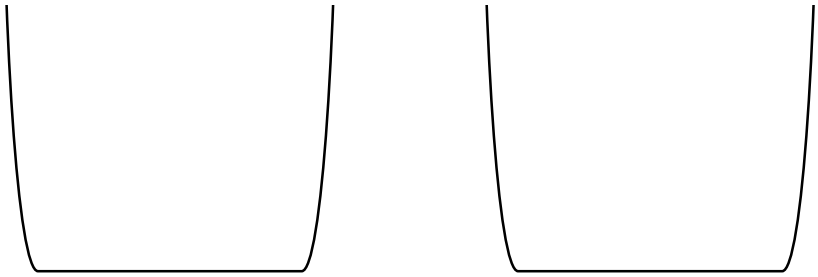


Figure 63
Example of Behavior Diagram Syntax

Using a partition name implies that the method is not in the partition that owns this behavior diagram, and that the method has been exported (i.e., is public) from the partition in which it is defined (Figure 64).

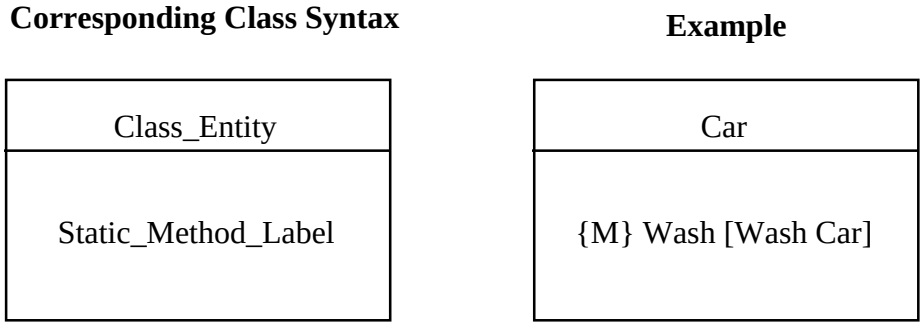


Figure 64
Behavior Diagram with a Partition

IDEF4 reuses the inheritance diagram syntax, including the external and internal specialization notation. To help differentiate behavior diagrams from inheritance diagrams, we often draw behavior diagrams from left to right instead of from top to bottom (Figure 65).

Figure 65
Behavior Diagram Including External Specialization

External specialization means refining the general behavior (i.e., the behavior of the external interface).

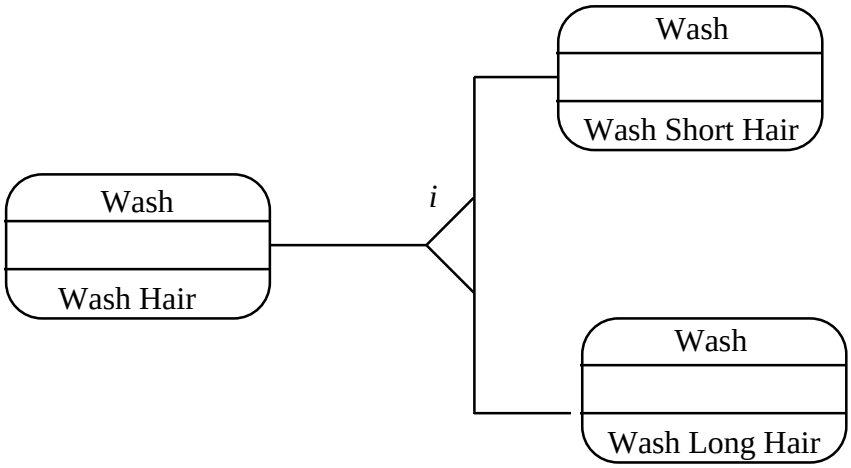


Figure 66
Behavior Diagram Including Internal Specialization

Internal specialization means refining implementations of the behavior (Figure 67).

Figure 67
Behavior Diagram Including Internal and External Specializations

The name of the behavior diagram is the system or partition behavior that is being described (i.e., the behavior family, which is really the message name).

Classes and Methods

If a method has no class(es) associated with it, it is not meant to be implemented – it is used to capture a behavioral specification that will be further specified.

If a method has exactly one class associated with it, the method is considered to be owned by the class and is part of the class's encapsulation.

If a method has more than one class associated with it, the method is a multi-method. It does not belong to any of the specified classes, although each class involved will have a feature that references the method. It belongs to the partition in which it is specified (or globally if there is no partition).

Behavior Specification

The behavior specification of a method is defined on a behavior specification form (Table 7). The behavior specification form contains the following elements:

Method name – unique name for the method (within the partition).

Behavior/Message Name – name of the Behavior Diagram that this method resides on. All methods reside on some Behavior Diagram.

Partition – the partition that the method is in; it could be the system (global) partition.

Classes – list of classes that participates in the method. It can have zero, one, or many.

Protocol – the input and output parameters and their types of the method; not to be confused with the classes that specialize this method.

Specialization – the parent and child method specifications that inherit to or from this method on the MTD.

Definition/Description – an overview of the method.

External Specification – a set of constraints that say what the outside world can expect from this method (i.e., what and when).

Internal Specification – a set of constraints that say how the method's internals should work (i.e., how).

Table 7. Behavior Specification

Author	*J. Smith*	**Project**	*IICE*	**Date**	*6/1/94*	**Revision**	*3*
Method Specification	Name	Method Name	Behavior	Msg Name	Partition	Partition Name	
Description		Method Description					
External Specification		*Specification of external behavior of method*					
Input Parameters		*Input parameter specification*					
Output Parameters		*Output parameter specification*					
Parent Methods		Child Methods					
List of parent methods		*List of child methods*					
Internal Specification		*Specification of internal implementation details of method*					

DYNAMIC MODEL

Objects and relationships change over time. In IDEF4, we first define the static structure of the problem using the Static Model. The aspects of the IDEF4 design that are concerned with time and change are defined in the Dynamic Model. Once the objects and structural relationships have been identified, the time varying behavior must be defined. The Dynamic Model defines the dynamics of interaction among objects and the state change dynamics of objects. The Dynamic Model consists of a Client/Server diagram, for specifying inter-object communication, and a State Model, for specifying the behavior of individual objects in terms of states and state transitions.

Overview

IDEF4 uses events and messages to define communication between objects and to define transitions between the states of those objects. These are depicted using a Client/Server diagram to define the communication, and a state diagram to show the state transitions.

Terminology used to explain the Client/Server diagrams and state diagrams includes action/message, event, object state, object life cycle, and object communication.

State

A state is an abstraction of the attribute values of an object. Sets of values are grouped together in a state in accordance with properties that affect the behavior of an object. For example, an account is either in the black or in the red, depending on whether the amount of funds attributed to the account is positive or negative. In IDEF4, a state specifies the response to events. The response of a state to an event may be an action or a change of state.

Action/Message

Objects communicate by passing messages (and broadcasting events) to each other. Upon receiving a message, an object takes an action. The action is specified in a method. Normally the message name is the same as the method that is invoked. Actions are also performed when an instance transitions into a state as a result of some condition (event) being detected. For state models, the action is called out by a message which is associated with a message that is sent to the object when the event is detected. For example, a bank account can have the states of having a balance and being overdrawn (Figure 68) The labels before the messages are an abbreviated way of referring to the messages. If an account detects an event indicating that it has become negative (event), it should be marked as overdrawn (action). An action will occur because the message associated with the event is sent to the object.

Figure 68
State Diagram for a Bank Account

Event

An event is a signal that is broadcast by an object when a condition is detected condition at a point in time (i.e., a door being opened, a button being pressed, the departure of a flight, or an account transitioning into a negative balance). Objects are attuned to the event by associating a message to be received with the event. Zero or more objects may respond to the event. Events are often related. For example, the door opening event and the door closing event form a cycle. Responding to an event may cause a state transition in an object.

State Transitions

State transitions define how objects move from state to state. The graphical syntax to represent transitions in state is shown in Figure 69.

The chevron on the event/message relation indicates the direction of the communication. Shadowed event/message chevrons denote parameters being passed between participating objects and states. Dashed lines indicate asynchronous messages. Solid lines indicate synchronous messages. For synchronous messages, the client object waits for a reply from the server object. For asynchronous messages, the client object continues executing. As can be seen from Figure 69, the event is optional on the relationship. If the event is left out, then the message is issued directly from the calling object.

Figure 69
Event/Message Syntax

Object Life Cycle

An object life cycle is the set of states and transitions defined by events and actions. For example, bank accounts can exist in the states closed, overdrawn, or active. The events that are meaningful include open, close, withdraw, deposit, (become) positive, and (become) negative. The actions that are meaningful in these states are create, delete, debit, credit, deactivate, and activate. The object life cycle defines allowable ways to enter and exit its states using event/action pairs. Note that creation and deletion of objects involve start and end states.

Object Communications

Objects interact to perform the mission of a system. The transfer of funds from a savings account to a checking account requires communication between an authorized user, a transaction object, the checking account, and the savings account (Figure 70).

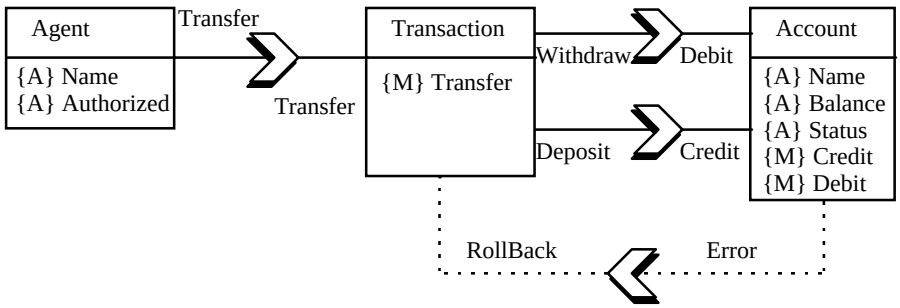


Figure 70
Classes Involved in Funds Transfer

An agent sends a «Transfer» message to a transaction object, which initiates a withdrawal and then a deposit to the different accounts. The transaction object ensures that all parts of the transaction complete; otherwise it rolls back. Savings account and checking account are subclasses of account, so both of them inherit the debit and credit methods from account. The transaction object waits for the withdrawal to complete before issuing the deposit. If the account cannot process the request, it stops the transaction and signals an error. The transaction object

uses the information passed to it by the error event to rollback transactions so that the accounts are returned to their original state.

The example in Figure 71 shows a simulation using instances. An authorized agent, Agent «[D. Brown],» initiates a transfer event on a transaction object.

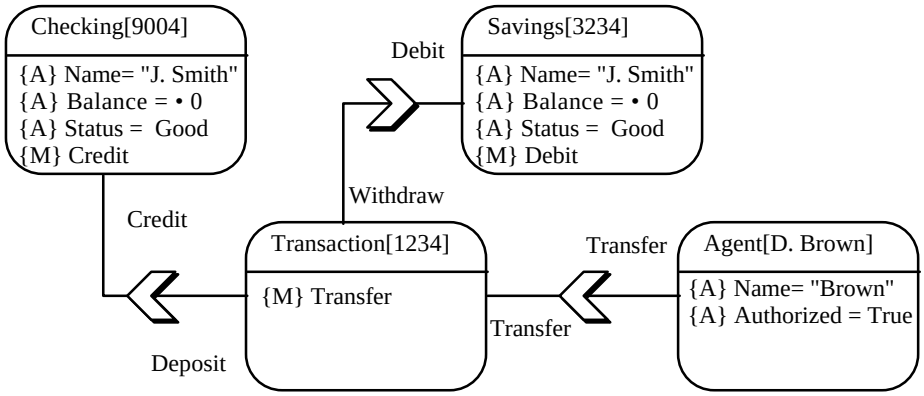


Figure 71
Instances Involved in Funds Transfer

Communication between objects occurs using a message and events. The action performed in response to a message is implemented by a method in an object. A method refers to the behavior of an object, stating what actions the object can perform. A message can be sent to other objects or it can be sent from an object to itself. An object life cycle is a set of states and transitions for a particular object. With a basic understanding of these elements, the designer can then begin creating Client/Server diagrams and state diagrams.

Client/Server Diagram

In IDEF4, the Client/Server diagrams are constructed first because they provide requirements input to the state diagrams. The state diagrams provide detailed descriptions of the behavior of each separate object in the system, whereas the Client/Server diagrams address the dynamics of the system as a whole.

Client/Server diagrams define coordinated behavior between state diagrams by defining the communication between objects in terms of event/message pairs. For example, in Figure 72, when a door is opened, the event «open» is generated, causing the message/action «start alarm» to be sent to an alarm system. If the event is left out of the diagram, then the message is issued directly from client object to server object.

Figure 72
Event/Message Communication Between Objects

Client/Server diagrams provide a graphical summary of event/message communication between state diagrams of objects in the system, external systems, external devices, and operators. An event that is generated by one state diagram and received and acted on by another is represented by an arrow link pointing from the object generating the event to the object receiving the event. The arrow is annotated with the event name and number, and the action/message name and number. Events that are generated and received by the same state diagram are omitted from the Client/Server diagram. If the partitions in the client server diagrams are tagged with network node designators, then the physical partitioning of the generated software is specified.

The designer will go through four steps to create a Client/Server diagram. These steps are shown in the following list:

(1) Identify compatible components,

(2) Assemble components,

(3) Simulate Client/Server diagram, and

(4) Develop state diagram.

These steps will be discussed using the example of a track event.

Example: A Track Event

Track and field meetings provide a wealth of examples of the interplay between events and actions in the real world. Figure 73 shows a sprint event partition that has the starting judge and runner classes imbedded in it.

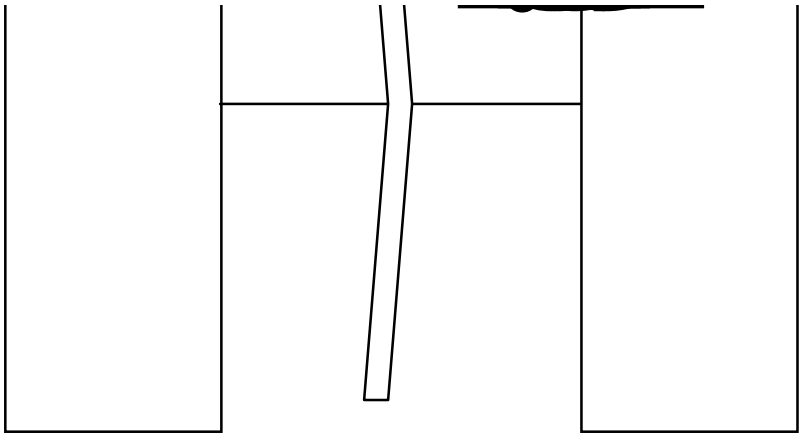


Figure 73
Track Race Scenario

The starting judge issues commands and the runners respond to these commands in a predetermined way. When the starting judge issues the «on your marks» command (shown in Figure 73 as the Marks event), the runners respond by getting into position. The starter then issues the «get set» command (the Get_Set event) and the runners respond by moving into the crouched position ready to start. The starting judge then fires the starting pistol and the runners start running. If a runner false starts, the starting judge fires the pistol a second time.

Identify Compatible Components

The first step in the development of the Client/Server diagram is to identify the components in the partition. The practitioner should try to select ready built design components before identifying new components to be designed and built.

Assemble Components

The next step is to assemble the components by defining the communication relations between objects. This defines the interfaces between the objects. For Server Database and End-user Database partitions, this would involve defining all of the messages that would be sent between the partitions.

Simulate Client/Server Model

In step three, the practitioner tests the Client/Server diagram by simulating event/action occurrences. The state diagrams for all components must be available in order to perform the simulation. Problems identified in the simulation of the Client/Server diagram will lead to selection of different components, or altering the state diagram of existing components.

Develop State Diagram

In order to fully specify the Client/Server model, the state diagram will have to be defined for each object. The track meet example gives the designer a very simple situation for building a Client/Server diagram. The next section discusses the incorporation of COTS or legacy software code into the Client/Server diagram.

Client/Server Encapsulation of Legacy and Commercial Systems

Often the design needs to use existing system components. Most commercial software applications have programmer interfaces, messaging interfaces, or object-level interchanges to allow the applications to be incorporated in system solutions. For example, it is common to use a commercial database system to store data, a spreadsheet to analyze the data and generate charts, and a word processor to print reports on the analysis. Protocols such as dynamic data exchange (DDE™) for messaging, object linking and embedding (OLE™) for exchanging data , and open database connectivity (ODBC™) for interfacing to databases make it possible to integrate commercial off the shelf (COTS) software. These protocols form part of what is known as middle ware – software that enables interapplication integration.

IDEF4 uses the partition to encapsulate the COTS software by defining the messages that the software responds to and the events that the software can generate. There are three levels of COTS software: (1) full-blown applications like spreadsheets and word processors, (2) applets or mid-level applications like spell checkers, grammar checkers, and print pre-viewers, and (3) executable software components such as buttons, input fields, and forms.

Figure 74 shows a Client/Server diagram for a partition containing a commercial word processor and a custom application that generates a report using the word processor. The custom application is a partition that maintains information about widgets and uses a form containing an input field and a button that are commercial components. The designer is responsible for designing the partition called *My-Application* which will use these components.

Figure 74
Design For COTS Client/Server System

The example in Figure 74 shows how the IDEF4 notation deals with COTS seamlessly. Only the external interface of the COTS product must be specified; the internals are the responsibility of the software manufacturer. This approach reduces the amount of code that will need to be maintained in the implemented system, and shifts the burden of maintenance from code maintenance to component configuration management. For legacy systems, a partition is used to encapsulate the legacy code so that wrappers can be designed to enable the software to integrate with other design components. This encapsulation contains the actions and events that other design components will need to interface with the legacy software.

State Diagram

Each object in the system has a state diagram which can be formalized as a set of states and event/action transitions (Figure 75). A state represents a situation or condition of the object during which certain constraints apply. The constraints may be physical laws, rules, and policies. Events are triggers that cause the object to initiate a transition from one state to another. An action occurs when the object enters the new state. The action is a message associated with a method that the object executes on arrival in the state.

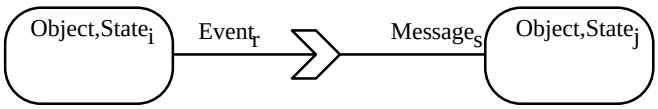


Figure 75
Object State Communication

IDEF4 uses a specialized representation for the situations when objects first come into existence (creation) and when they cease to exist or cease to participate in dynamic behavior (final). IDEF4 further distinguishes between events that are generated internally by the object in the state diagram and objects communicating with the state diagram. In the state diagram, external events are shown in parenthesis to indicate that they are place holders or variables referencing events generated externally. De-referencing the external events in the state diagram preserves encapsulation, while allowing the state diagram to reference external events without defining them. This technique makes the design objects extremely re-usable.

The state diagram can be created using a process similar to the process for creating a Client/Server diagram:

(1) Identify states,

(2) Identify internal events,

(3) Identify kinds of external events,

(4) Identify action, and

(5) Simulate state diagram.

The development of the state diagram will be discussed using a banking example which walks through these five steps and culminates, in an additional step, with code generation.

Banking Example

A banking example illustrates the subtleties of the state diagram. One of the most important objects in a banking system is the account. The state diagram for Account is shown in Figure 76. The following paragraphs will detail the process for building this Account State Diagram.

In creating the state diagram, it is important to acquire a detailed description of the dynamics of the account and the requirements placed on the account. Interviews with the customer and Client/Server diagrams using the account object provide input on required behavior.

Figure 76
State Diagram For a Bank Account

Step 1: Identify States

The required account behavior has been simplified for this example. An account can be active, overdrawn, or closed. An account may be created with the client's name and zero balance. When the account is active, amounts may be debited on withdrawals and credited on deposits until the balance goes negative, at which time it becomes overdrawn. When the account is overdrawn, amounts may be credited on deposit until such time as the balance becomes positive. When an overdrawn account becomes positive, it is activated. In the event that the account is closed, the account is deactivated for a period, then deleted from the system.

A state diagram must be designed so that it unambiguously specifies the behavior of the account. The first step in designing the account state diagram is to identify a set of suitable state abstractions. Clearly, the account is either closed, active, or overdrawn. These situations can be abstracted as three states plus the start and final states:

(S_0) **start** before account is created,

(S_1) **active** normally operating account,

(S_2) **overdrawn** account,

(S_3) **closed** account, and

(S_f) **final**: account removed from system.

The object state box contains a state title consisting of the state ID, the classname, and the state name. The body of the state box contains the attribute/value pairs that define the state. It is often convenient to define the attribute value as a range using a constraint, (for example «balance

> 0»). If the effect of each action on the attributes is defined, then code can automatically be generated from the state diagram. The states can be defined as follows:

(S_1) **active** {Status == Active},

(S_2) **overdrawn** {Status == Overdrawn}, and

(S_3) **closed** {Status == Closed}.

This information is shown in Table 8, the Account State Specification Form. The table contains the state ID, name, description, condition, and action.

Table 8. IDEF4 Account State Specification Form

State Specification Form	Class Name(s): Account			
ID	**Name**	**Description**	**Condition**	**Entry Action**
S_0	Start	No account instance exists.	None	None
S_1	Active	Normal operating mode off Account. Can accept debits and credits.	Status==Active	Status:=Active
S_2	Overdrawn	Restricted operating mode of Account. Only credits may be applied to this account.	Status==Overdrawn	Status:=Overdrawn
S_3	Closed	Account has been closed and balance paid out, awaiting deletion.	Status==Closed	Status:=Closed
S_F	Final	Account removed from system.	None	None

Step 2: Identify Internal Events

The next step is to identify the events that are triggered by the account. These events are easily identified by examining situations that cause state changes. These situations are as follows:

(E_5) balance of active account becoming **negative**, and

(E_6) balance of an overdrawn account becoming **positive**.

The internally generated events can be defined as follows:

(E_5) **negative** {Balance < 0 };

(E_6) **positive**{Balance ≥ 0 };

This information is shown in the Account State Specification Form (Table 9). The Table contains the event ID, name, description, and condition.

Table 9. Internal Event Specification Form for Account

Internal Event Specification Form		Class Name: Account	
ID	**Name**	**Description**	**Condition**
E_5	**negative**	Balance of active account becoming negative	S_1 and Balance < 0
E_6	**positive**	Balance of an overdrawn account becoming positive	S_2 and Balance ≥ 0

Step 3: Identify Kinds of External Events

Numerous kinds of events are generated by other objects that trigger state changes in state diagrams. External events are added to the state diagram as referential attributes that can be bound to external events when the state diagram is used, such as in a Client/Server diagram. This ensures that encapsulation of the account object is not compromised, and can be reused in different partitions. The references to external events are shown in parenthesis on the state diagram (Figure 76) and are as follows:

(E_1) **(open)** account triggered by an external object (e.g., a teller);

(E_2) **(close)** account triggered by an external object (e.g., a teller),

(E_3) **(withdraw)** triggered by an external object (e.g., a teller or automated teller machine [ATM]),

(E_4) **(deposit)** triggered by an external object (e.g., a teller or ATM), and

(E_7) **(remove)** triggered by an external object (e.g., a system administrator).

This information is shown in the Account External Event Specification Form (Table 10). The Table contains the external event ID, name, and description.

Table 10. External Event Specification Form for Account

External Event Specification Form		Class Name: Account
ID	**Name**	**Description**
E_1	**open**	event for starting a new account by an external object (e.g., a teller)
E_2	**close**	event for closing an account triggered by an external object (e.g., a teller)
E_3	**withdraw**	event for withdrawing money from an account triggered by an external object (e.g., a teller or ATM)
E_4	**deposit**	event for depositing money into an account triggered by an external object (e.g., a teller or ATM)
E_7	**remove**	event for expunging records from the system triggered by an external object (e.g., a system administrator)

Step 4: Identify Actions

After the events are identified, the actions performed by the account must be identified. The kinds of decisions that are being made here are as follows: «In state s_1, if event e_2 is generated, what state is entered, and what is the appropriate action?» If the balance becomes negative when the account is active, the account is set to overdrawn and a $20 amount is debited from the account. If the balance becomes positive when the account is overdrawn, the account is made active. Through this exercise, the following actions are identified:

(M_1) **create** new account,

(M_2) **deactivate** closed account,

(M_3) **debit** amount from balance,

(M_4) **credit** amount to balance,

(M_5) **overdraw** account,

(M_6) **activate** overdrawn account, and

(M_7) **delete** closed account.

As one enumerates all possible connections between states using events generated and appropriate responses, one soon realizes the need to use a state diagram to manage the complexity of the problem. The state diagram contains object states linked by dynamic relations defined by event/action pairs (Figure 77).

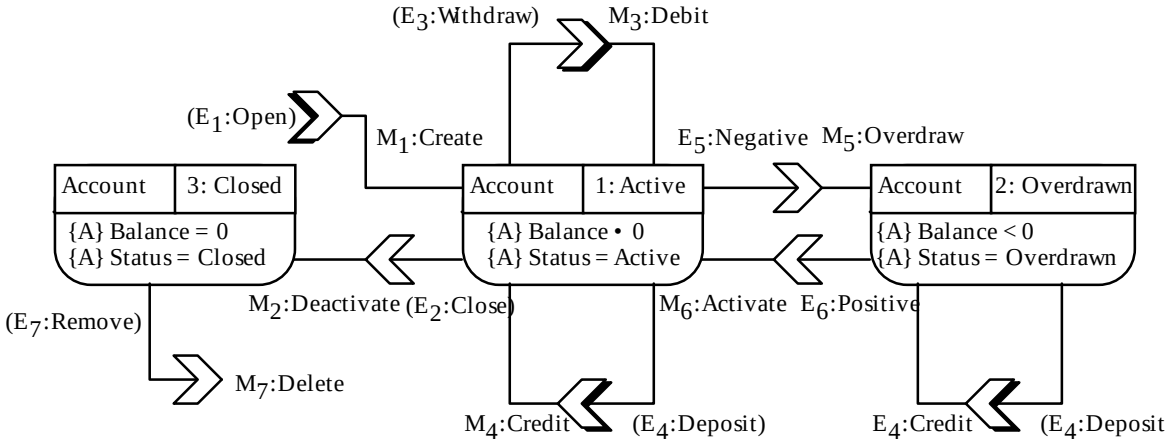


Figure 77
State Diagram for a Bank Account

The arrow on the dynamic relation indicates the causality of the event/action pair. Events and actions can be abbreviated for easier reference.

The semantics of the actions performed by the account can be defined as follows:

(M_1) **create** (Name){Balance = 0; Owner = Name},

(M_2) **deactivate** (){debit (Balance)},

(M_3) **debit** (Amount){Balance = Balance - Amount;},

(M_4) **credit** (Amount){Balance = Balance + Amount;},

(M_5) **overdraw** (){Balance = Balance - Penalty;},

(M_6) **activate** (){}, and

(M_7) **delete** (){Delete;}.

This information is shown in the Account Action Specification Form (Table 11). The table contains the action ID, name, description, and action description.

Table 11. Action Specification Form for Account

Action Specification Form		Class Name: Account	
ID	**Name**	**Description**	**Specification**
M_1	**create**	Start new account	**create** (Name) {Balance = 0; Owner = Name}
M_2	**deactivate**	Close account	**deactivate**() {debit (Balance)}
M_3	**debit**	Subtract amount from balance	**debit** (Amount) {Balance = Balance - Amount;}
M_4	**credit**	Add amount to balance	**credit** (Amount) {Balance = Balance + Amount;}
M_5	**overdraw**	Flag account as overdrawn	**overdraw**() {Balance = Balance - Penalty;}
M_6	**activate**	Reactivate overdrawn account	**activate** (){}
M_7	**delete**	Remove account from system	**delete** (){Delete;}

After gathering information for the state diagram using the four step procedure, the designer should complete the state diagram.

Table 12. Object State Transition Matrix for Account

Object State Transition Matrix			Class Name: Account			
Event State	E_1 (Open)	E_2 (Close)	E_3 (Withdraw)	E_4 (Deposit)	E_5 Negative	E_6 Positive	E_7 (Delete)
S_0:Start	1:M_1	Ø	Ø	Ø	Ø	Ø	Ø
S_1:Active	Ø	3:M_2	1:M_3	1:M_4	2:M_5	Ø	Ø
S_2:Overdrawn	Ø	Ø	Ø	2:M_4	Ø	1:M_6	Ø
S_3:Closed	Ø	Ø	Ø	Ø	Ø	Ø	FM_7
S_F:Final	Ø	Ø	Ø	Ø	Ø	Ø	Ø

Table 12 shows the relations between states, events, and actions in a matrix form. The rows are identified by states and the columns by events. Each cell in the matrix identifies the state to transition to and the action initiated upon entry into that state.

Step 5: Simulate State diagram

In this step, the practitioner tests the state diagram by simulating event/action occurrences against different states. Problems in the state diagram may lead to state diagram design changes and ultimately to changes in the Client/Server diagram.

Step 6: Generate Code

Given the state diagram and the definitions of the states, internal events, and actions, the source code for the state diagram can be generated automatically. For a state transition (S_u S_v) defined by the event/action pair (E_i M_j), the code will take on the form:

If S_u and E_i then do {S_v ;M_j)

Thus, when the account state transitions from active to overdrawn, the code fragment generated would be:

If S_1 and E_5 then do {S_2; M_5;} which expands to:

```
If (status == active and balance < 0) then
      do      {       status = overdrawn;
                      Balance = Balance - 20;
              }
```

For (S_1, S_1)(S_2, S_2) (E_4, M_4) the following pseudo code could be generated

```
void account::credit(real amount)

{
      if (or S1 S2) balance = (balance + amount);
}
```

Oven Example

The following example depicts the creation of a Client/Server diagram and state diagram for a microwave oven with a simple control interface (Figure 78).

Figure 78
A Microwave Oven with a Simple Interface

The oven has a door, an «Add Minute» button, a «Clear» button, and an «Open» button. To operate the microwave, the door is opened by pressing the «Open» button, food is placed in the microwave, the door is closed, then the «Add Minute» button is pressed repeatedly until the desired amount of time in minutes is reached. The microwave begins cooking as soon as any amount of time is entered, and stops cooking when the door is opened. The microwave resumes cooking when the door is closed, if there is still time left. The «Clear» button can be used to reset the microwave. The microwave has a software controller that ensures safe operation. The controller uses a software clock to track the seconds.

Example of Client/Server Diagram: Oven System

All four of the steps involved in creating a Client/Server diagram will be used in this example.

Step 1: Identify Compatible Components

Only the external interfaces of the components of the microwave are visible to one another. These components must be connected to each other using events and messages. For example, the Power_On and Power_Off events should be connected to a device that cycles power on and off. The AddOne message should be connected to an input device that allows users to indicate cooking time. Figure 79 shows the IDEF4 design partition for the microwave. The Door, AddButton, and ClearButton objects are real-world (®) objects that are public to the partition because they are accessible to users of the microwave. The controller, timer, and power switch objects are private because they are not accessible to users.

Figure 79
IDEF4 Design Partition For Microwave

Figure 80 shows the IDEF4 components of the microwave. The software part of the system consists of a controller and clock. The external interfaces consist of a power switch, «Clear» button, «Add» button, and Door.

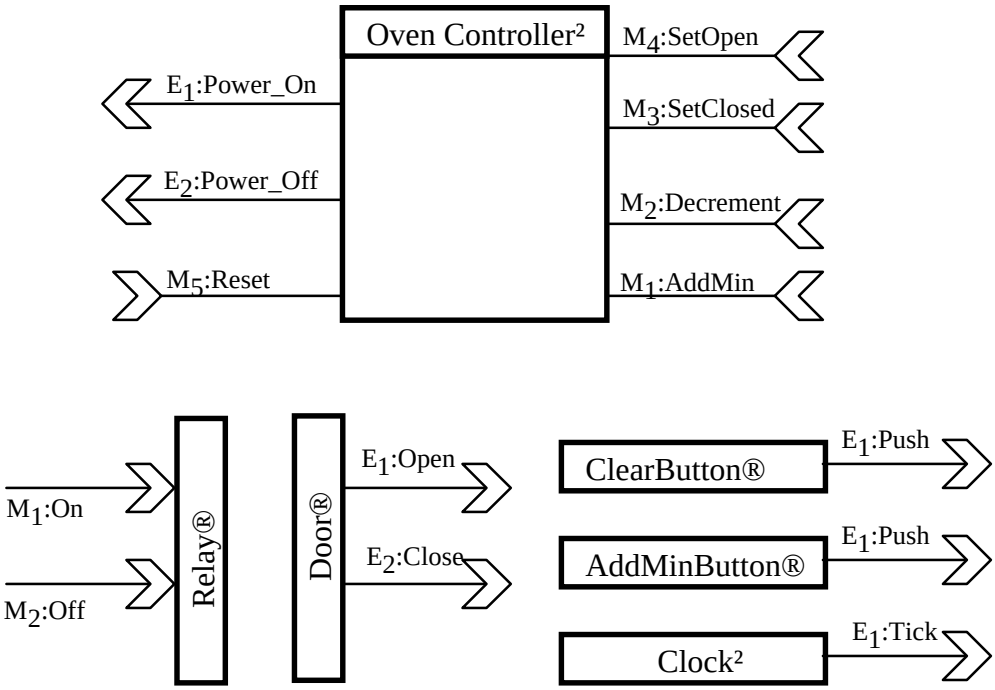


Figure 80
Components in Microwave Example

For example, the controller generates Power_On and Power_Off events and performs actions such as Add Minute, decrement counter, and reset. These components are assembled by linking an event generated by one object to an action that another object may perform.

Step 2: Assemble Components

Figure 81 is an IDEF4 Client/Server diagram showing how the microwave components communicate using events and messages. The *ClearButton to Microwave Controller* link may be read as *when the ClearButton is pressed, a push event is generated which causes the reset message to be sent to the oven controller which performs the reset action.*

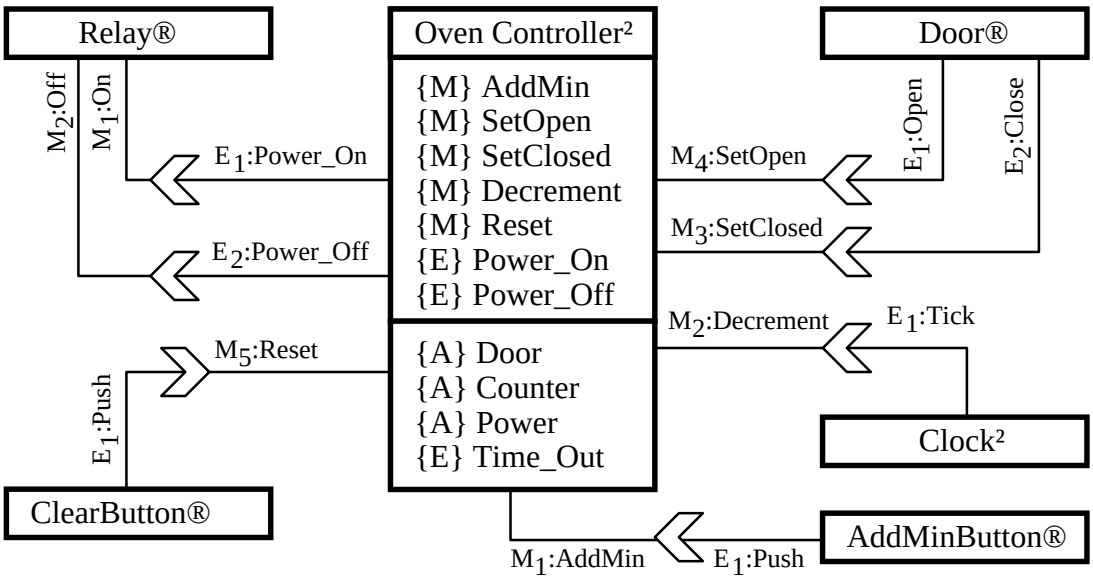


Figure 81
Oven Controller Client/Server Diagram

The methods, events, and attributes of the controller have been added to the controller object. The door, counter, and power attributes of the controller are private so they cannot be seen by the objects external to the controller. The Time_Out event is also private because it is only used internally by the controller. The oven Client/Server matrix (Table 13) summarizes the Client/Server relations in a convenient tabular form.

Table 13. Oven Client/Server Communications Matrix

Client/Server Matrix			Partition Name: Microwave Oven			
Server / Client	Controller	PowerSwitch	ClearButton	AddMinButton	Clock	Door
Controller	Ø	E_1:M_1 E_2:M_2	Ø	Ø	Ø	Ø
PowerSwitch	Ø	Ø	Ø	Ø	Ø	Ø
ClearButton	E_1:M_5	Ø	Ø	Ø	Ø	Ø
AddMinButton	E_1:M_1	Ø	Ø	Ø	Ø	Ø
Clock	E_1:M_2	Ø	Ø	Ø	Ø	Ø
Door	E_1:M_4 E_2:M_3	Ø	Ø	Ø	Ø	Ø

Step 3: Simulate Client/Server Diagram

In step three, the practitioner tests the Client/Server diagram by simulating event/action occurrences against the diagram. The state diagrams for all components must be available in order to perform simulation. Problems identified in the simulation of the Client/Server diagram indicate that different components should be chosen, or existing components in the Object State Model should be altered.

Step 4: Generate Code

The interaction between the clock and controller objects can be used to illustrate code generation at the system level. In the state diagrams for clock, there is a method called Wait that pauses for a second, then generates the Tick event. The controller state diagram has a decrement method that subtracts one unit from a counter. The methods could be specified as follows:

Wait() {Pause(1); event(Tick);}

Decrement() {counter = counter - 1; if counter = 0 then event (Time_Out);}

The Tick/Decrement link between clock and controller causes the controller to perform a decrement action every time the Tick event is generated.

Example of State Diagram: Microwave System

We continue the microwave example used to illustrate the Client/Server diagram to illustrate the state diagram (Figure 82).

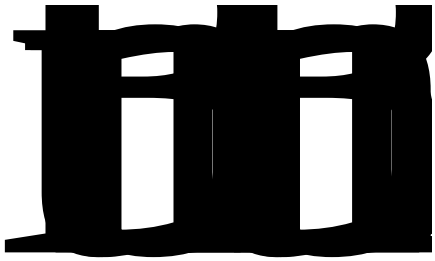


Figure 82
Oven with a Simple Interface

As explained previously, the microwave has a door, an «Add Minute» button, and an «Open» button. To operate the microwave, the door is opened by pressing the «Open» button, food is placed in the microwave, the door is closed, then the «Add Minute» button is pressed repeatedly until the desired amount of time in minutes is reached. The microwave begins cooking as soon as any amount of time is entered and stops cooking when the door is opened. The microwave resumes cooking when the door is closed, if there is still time left. The following paragraphs detail the process for building a state diagram of the microwave's controller.

In creating the state diagram, it is important to acquire a detailed description of the dynamics of the account and the requirements placed on the account. Interviews with the customer and Client/Server diagrams using the controller object provide input on required behavior.

Step 1: Identify States

The first step in designing the controller's state machine is to identify a set of suitable state abstractions. Clearly, the controller must be aware of the door open or closed status, the time on the clock, and the power on or off status. These situations can be abstracted as four states:

(S_1) the oven door is **open**, **no time** on the clock, and power **off**,

(S_2) the oven door is **open**, **time** on the clock, and power **off**,

(S_3) the oven door is **closed**, **time** on the clock, and power **on**, and

(S_4) the oven door is **closed**, **no time** on the clock, and power **off**.

The object state contains a state title consisting of classname and state name. The body of the state box contains the attributes/value pairs that define the state. It is often convenient to define the attribute value as a range using a constraint, for example «counter > 0.» If the effect of

each action on the attributes is defined, code can automatically be generated from the state diagram

One internal event, (namely time out) is specified by the counter reaching zero when the microwave is operating. The power, and counter attributes can be used to uniquely identify the states of the controller:

(S_1) (door == **open** and **counter** = 0),

(S_2) (door == **open** and **counter** > 0),

(S_3) (door == **closed** and **counter** > 0), and

(S_4) (door == **closed** and **counter** = 0).

IDEF4 requires that each state identified should have: (1) an identifier that is uniquely identifies it in the state diagram, (2) a descriptive name, (3) a textual description of the state, (4) a condition for identifying the state in terms of attribute values, and (5) a set of actions that need to be performed by all transitions into the state. This information is shown in the Controller State Specification Form (Table 14). The table contains the state ID, name, description, condition, and action.

Table 14. IDEF4 Controller Object State Specification Form

Object State Specification Form		Class Name(s): Controller		
ID	**Name**	**Description**	**Condition**	**Entry Action**
S_1	Open No Time Off	the oven door is **open**, **no time** on the clock, and power **off**	(door == open and counter = 0)	
S_2	Open Time Off	the oven door is **open**, **time** on the clock, and power **off**	(door == open and counter > 0)	
S_3	Closed time on	the oven door is **closed**, **time** on the clock , and power **on**	(door == closed and counter > 0)	SetPowerOn
S_4	Closed time off	the oven door is **closed**, **no time** on the clock , and power **off**	(door == closed and counter = 0)	SetPowerOff

Step 2: Identify Internal Events

The next step is to identify the events that are triggered by the controller. These events are easily identified by examining a situation that causes controller state changes:

(E_4) the **time-out** event generated by the controller's counter reaching zero

The internally generated events can be defined as follows:

(E_4) **time-out** {Counter = 0;}

IDEF4 requires that each object-internal event identified should have: (1) an identifier that uniquely identifies it in the state diagram, (2) a descriptive name, (3) a textual description of the event, and (4) a condition for triggering the event in terms of object attribute values. This information is shown in the Controller State Specification Form (Table 15). The table contains the event ID, name, description, and condition.

Table 15. Internal Event Specification Form for Controller

Internal Event Specification Form		Class Name: Controller	
ID	**Name**	**Description**	**Condition**
E_4	**time-out**	the time-out event generated by the controller's counter reaching zero	{Counter = 0;}

Step 3: Identify Kinds of External Events

The next step is to identify the kinds of external events that could trigger state changes in the control system. These events are easily identified by looking at the external interface of the oven:

(E_1) the **close** event generated from the door closing (external),

(E_2) the **open** event generated from the door opening (external),

(E_3) the **buttonpressed** event generated by «AddMinute» (external),

(E_5) the **tick** of the microwave clock (external), and

(E_6) the **clear** event generated by the clear button (external).

IDEF4 requires that each **kind** of external event identified should have: (1) an identifier that uniquely identifies it in the state diagram, (2) a descriptive name, and (3) a textual description of the kind of event. This information is shown in the Controller External Event Specification Form (Table 16). The table contains the external event ID, and name, and description.

Table 16. Internal Event Specification Form for Controller

External Event Specification Form		Class Name: Controller
ID	**Name**	**Description**
E_1	**close**	the **close** event generated from the door closing (external)
E_2	**open**	the **open** event generated from the door opening (external)
E_3	**buttonpressed**	the **buttonpressed** event generated by «AddMinute» (external)
E_5	**tick**	the **tick** of the microwave clock (external)
E_6	**clear**	the **clear** event generated by the clear button (external)

Step 4: Identify Actions

After the events are identified, the actions the controller needs to respond with must be identified. The kinds of decisions that are being made here are as follows: «In state s_1, if event e_2 is generated, what state is entered, and what is the appropriate action?» If the door is opened when the microwave is operating, operation must be interrupted. If the door is opened when the microwave is not operating, operation must be prevented from starting until the door is closed. Whenever the «AddMinute» button is pressed, a minute must be added to the microwave's timer. Through this exercise, the following actions are identified:

(M_1) **add minute** to counter,

(M_2) **subtract second** from counter,

(M_3) **set open** controller,

(M_4) **set closed** controller, and

(M_5) **reset** the controller.

The semantics of the actions performed by the account can be defined as follows:

(M_1) **add_min**{counter = counter + 60;}

(M_2) **decrement**{counter = counter -1;

if counter = 0 then event(time-out)}

(M_3) **set_open**: door = open;

(M_4) **set_closed**: door = closed;

(M_5) **reset:** counter = 0;

(M_6) **SetPowerOff:** power = off; event(Power_Off)

(M_7) **SetPowerOn:** power = on; event(Power_On)

IDEF4 requires that each action identified should have: (1) an identifier that uniquely identifies it in the state diagram, (2) a descriptive name, (3) a textual description of the event, and (4) an action description. This information is shown in the Controller Action Specification Form (Table 17). The table contains the action ID, name, description, and action description.

Table 17. Action Specification Form for Controller

Action Specification Form		Class Name: Controller	
ID	**Name**	**Description**	**Specification**
M_1	add_min	add minute to counter	add_min() {counter = counter + 60;}
M_2	decrement	subtract second from counter	decrement() {counter = counter -1; if counter = 0 then event(time-out)}
M_3	set_open	set open controller	set_open() {door = open;}
M_4	set_closed	set closed controller	set_closed() {door = closed; }
M_5	reset	reset the controller	reset() { counter = 0; }
M_6	SetPowerOff	Set the power attribute to off and generate a power_off event.	SetPowerOff() { power = off; event(Power_Off);}
M_7	SetPowerOn	Set the power attribute to on and generate a power_on event.	SetPowerOn:() { power = on; event(Power_On)}

Step 5: Complete State Diagram

As one enumerates all the possible connections between states using events generated and appropriate responses, one soon realizes the need to use a state diagram to manage the complexity of the problem. The state diagram contains object states linked by dynamic relations defined by event/action pairs (Figure 83).

Figure 83
Oven Controller State Diagram

The arrow on the dynamic relation indicates the causality of the event/action pair. Events and actions can be abbreviated for easier reference.

Table 18 shows the relations between states, events, and actions in a matrix form. The rows are identified by states and the columns by events. Each cell in the matrix identifies the state to transition to and the action initiated upon entry into that state.

Table 18. Object State Transition Matrix for Controller

Object State Transition Matrix			Class Name: Controller			
Event State	E_1 (Close)	E_2 (Open)	E_3 (buttonpress)	E_4 time-out	E_5 (tick)	E_6 (clear)
S_1	S_4:M_3	Ø	S_2:M_1	1:M_4	Ø	Ø
S_2	S_3:M_3	Ø	S_2:M_1	2:M_4	Ø	S_1:M_5
S_3	Ø	S_1:M_4	S_3:M_1	Ø	S_3:M_2	S_4:M_5

S_4	Ø	S_2:M_4	S_3:M_1	Ø	Ø	Ø

Step 6: Simulate State diagram

In this step, the practitioner tests the state diagram by simulating event/action occurrences against different states. Problems in the state diagram may lead to state diagram design changes and ultimately to changes in the Client/Server diagram.

Step 7: Generate Code

Code can easily be generated from these specifications (i.e., the ticking of the external clock causes the counter to decrement when the microwave is operating).

if S_4 and E_5 then do {S_4; M_8} which expands to

if **E_5 and** (door == **closed** and power == **on)** **then** decrement (Self);

E_5 has not been bound to any external events because we have not specified the object(s) that generate events that are attached to this message.

DESIGN RATIONALE COMPONENT

The purpose of the design rationale component is to facilitate the acquisition, representation, and manipulation of the design rationale utilized in the development of IDEF4 Designs. The term rationale is interpreted as the reason, justification, underlying motivation, or excuse that moved the designer to select or adopt a particular strategy or design feature. More simply, rationale is interpreted as the answer to the question «Why is this design being done in this manner?» Most object-oriented design methods focus on the what the design is (i.e., on the final product, rather than why the design is the way it is).

The scope of the rationale component covers all phases of the IDEF4 development process, from initial conceptualization through both preliminary and detailed design activities.

Design rationale becomes important when there are options, that is, when a design decision is not completely determined by the constraints of the situation. Thus, (1) decision points must be identified, (2) the situations and constraints associated with those decision points must be defined, and (3) if options exist, the rationale both for the chosen option and for the reasoning for discarding other options (i.e., those design options not chosen) must be recorded.

Motivation

Supporting the evolution of integrated information systems with life cycles that could possibly extend over many career periods requires the explicit capture and storage of design rationale. In addition, capture of design rationale is also important during the development phase of large scale systems. In these situations, the logic (i.e., the chain of arguments or reasons) behind the design is invaluable to the down-stream developers, testers, and integrators.

Computer-aided software engineering (CASE) environments attempt to bring automated support to the design stage. In specific situations they have been demonstrated to accomplish the purpose for which they were intended. Even so, existing CASE tools are inherently limited in at least two important respects. CASE tools are intended to document various aspects of *what* a design is, but they were never intended to document *why* a design exists, certainly not in any methodical way.

Even when design rationale comments exist, they are just that – unstructured textual comments. Often, the only coherent record of the design rationale for a software system is distributed across many people, and at any one time, parts of it will have been forgotten or made unavailable.

The loss of design rationale results in repeating past mistakes and making decisions contrary to the original design assumptions. Furthermore, each problem that is corrected without design rationale causes other problems elsewhere in the design.

One benefit of maintaining design rationale is to force a statement of goals, as well as assumptions. Goals, like assumptions, are frequently not stated. Forcing their statement for the

purpose of rationale capture leads to a more focused, disciplined approach to design. Much of design thinking appears to be abductive in nature, with experience-directed insights being fashioned and rationalized in the context of the current task. This rationalization may be the only basis for understanding why a system is the way it is. Without the capture of this chain of reasoning or arguments, communication of the design becomes difficult and error prone.

Another motivation for rationale capture comes from the fact that the definition of subsystems and subsystem boundaries is an experimentation process, in which each designer discovers the boundaries he finally imposes. If the organization is to avoid costly errors, it must have knowledge of the paths of inquiry that failed, the path to the final success, and the final result.

Nature of Design Rationale

Considering the nature of design rationale (why and how), we need to contrast it with the related notions of: (1) design specification (what), and (2) design history (steps taken). Design specifications describe what (intent) should be realized in the final physical artifact. Design rationale describes why the design specification is the way it is. This includes such information as principles and philosophy of operation, models of correct behavior, and models of how the artifact is intended to behave as it fails. The design process history records the steps that were taken, the plans and expectations that led up to these steps, and the results of each step.

Design Rationale Phenomena

A general characterization of design rationale can be given as: «The beliefs and facts as well as their organization that the human uses to make (or justify) design commitments and to propagate those commitments.»

In our investigation into the nature of object-oriented design rationale, we have characterized both «types» of design rationale and «mechanisms» for representation of these types. Types of design rationale include arguments based on the following:

(1) The philosophy of a design, including:

   (a) Process descriptions of intended system operation, and

   (b) Design themes expressed in terms of particular object types standing in some specific relation or exhibiting some specific behavior.

(2) Design limitations expressed as:

   (a) range restrictions on system parameters, and

   (b) environmental factors.

(3) Factors considered in tradeoff decisions including:

(a) level of requirements matching,

(b) project budget or timing constraints,

(c) general method of doing business in the organization,

(d) technology available to realize (implement) the design, and

(e) technology available to test the resulting product.

(4) Design goals expressed in terms of:

(a) Use or lack of use of particular components,

(b) Achievement of particular structural arrangements,

(c) Priorities on problems requirements,

(d) Product life cycle characteristics (e.g., disposable versus maintainable, robustness, flexibility), and

(e) Design rules followed in problem or solution space partitioning, test/model data interpretation, or system structuring.

(5) Precedence or historical proof of viability.

(6) Legislative, social, professional society, business, or personal evaluation factors or constraints.

Possibly due to the commonness of routine design or the complexity of design rationale expression, the most common rationale given for a design is that it was the design that worked in a previous situation. Without making judgment on this phenomena, a minimum requirement for a design knowledge management capability is that it must be able to record historical precedence, as well as statements of beliefs and rationalizations for why a current design situation is identical to the one the previous design serviced. This phenomena may be less common in software design rationale. The malleability of object-oriented design gives rise to the belief that we can be creative each time. Note that in software, as contrasted with hardware, creating a new model based on last year's doesn't mean the same thing. In software, the reused parts are literally copied whole, not rebuilt from the same plans, so you lose the opportunity to fix small design flaws, and the interaction of the new parts with the old is likely to be much less well understood.

Another important rationale given for a design is that «it feels better,» or «it seems more balanced, symmetric.» There is an important aesthetic side to software design.

Finally, software design rationale includes expectations about how the design will evolve through the development process itself. For example, expectations might be had about how the program structure will probably change—note such expectations do not appear to be as well defined as similar expectations we have seen in mechanical hardware design.

The important general conclusion that can be made about the nature of design rationale is that it takes the form of a trace of a reasoning process. This trace starts with the element of the design that is being justified and provides a set of supporting arguments that ultimately «ground» (terminate) to proven elements of the problem space or the design space. This chain of arguments may also terminate in previously rationalized design elements.

Design Rationale Concepts

IDEF4's approach to design rationale uses a strategy based on the relations between design situations or states. The strategy is to cast rationale as the formation of involvement relations between a designer in a particular decision making situation and one of a number of different «constraining» situation types. These constraining situations can include:

(1) Convention Constraints (e.g., historical precedence, societal [both marketing and professional]),

(2) Nomic Constraints (e.g., Laws of nature),

(3) Necessary Constraints (e.g., model based),

(4) Requirements Based Constraints (e.g., customer or contractual requirements),

(5) Goal Based Constraints (e.g., customer, project team or personal design goals), and

(6) Resource Constraints (e.g., time, skills, manpower, money).

The procedure followed in IDEF4's rationale component uses two phases: Phase I describes the problem and Phase II develops a solution strategy .

Phase I: Describe Problem

Problems with the current state of the design are normally identified during the simulation activity of the design cycle. In the simulation activity, use scenarios are used to validate and verify the design or implementation. In Phase I the designer describes problems that exist in the current design state by: (1) identifying problems, (2) identifying violated constraints (requirements, physical laws, norms, etc.), (3) identifying needs for problem solution, and (4) formulating goals and requirements for the subsequent design iteration.

Identify Problems

The designer identifies problems in the current design state by stepping through the use cases in the requirements model to validate that the design satisfies requirements and to verify that the design will function as intended. The designer records observations of problems which are symptoms or concerns about the current design state. A problem is an area in the design

needing improvement as indicated by one or more related observations. A symptom is an observation of an operational failure or undesirable condition in the existing design. A concern is an observation of an anticipated failure or undesirable condition in the existing design.

The requirements model consists of a function[15] model and several process[16] models that describe intended system usage. The function model constrains the scope of the partition and what activities should be supported by the partition. The process models describe use scenarios of how the activities occur. The activities in the function models call out[17] process models. These models map to requirements and goals. The function use model is used for validating and verifying interfaces and activities, and the process use model is used for validating and verifying process flow and logic.

Figure 84 depicts a design for a partition called Sys, showing its constituent static and dynamic models as well as its associated requirements model. The requirements model contains an IDEFØ function model whose activities call out IDEF3 process scenarios. The designer walks the design through these use cases to detect situations that have not been adequately supported.

[15]Function models may be expressed in IDEFØ, but should not contain decompositions by type.
[16]Process models may be expressed in IDEF3.
[17]IDEFØ activities call IDEF3 process scenarios using call mechanism arrows.

Figure 84
Static, Dynamic, and Requirements Models for Sys Partition

Identify Constraints

The designer then identifies the constraints that the problems violate or potentially violate. These constraints include requirements, goals, physical laws, conventions, assumptions, models, and resources. Because the activities and processes in the use case scenarios map to requirements and goals, the failure of the design in any use case activity or process can be traced directly to requirements statements and goal statements.

Figure 85 illustrates the mapping between the analysis model's function and use scenarios and the requirements and goal statements. When the design fails to adequately support activities and use scenarios, the requirements model allows the designer to easily identify the requirements constraints or goal statements that have been violated.

Figure 85
Functions and Use Scenarios Mapping to Requirements and Goals

Identify Needs

The designer then identifies the necessary conditions or needs for solving the problems. A need is a necessary condition that must be met if a particular problem or set of problems is to be solved. It may be necessary in the needs statement to show the necessity for relaxing requirements and goal constraints governing the design.

Formulate Goals and Requirements

Once the needs for the design transition have been identified, the designer formulates requirements that the solution must satisfy and goals that the solution should attempt to satisfy. A requirement is a constraint on either the functional, behavioral, physical, or method of development aspects of a solution. A design goal is a stated aim that the design structure and specifications are to support.

Phase II: Formulate Solution Strategies

Once the requirements and goals have been established, the design team formulates alternative strategies for exploration in the next major transition in the design.

One important aspect that distinguishes good designers is the ability to choose between making strategic design decisions and tactical design decisions. Strategic design decisions have sweeping architectural implications (i.e., the decision to use an OODBMS, the separation of responsibilities in client server architecture, the choice of mechanism for inter-process

communication). Tactical decisions represent the essential details that complete architectural decisions (i.e., the schema of a database, the protocol of a class, and the signature of a member function).

Design strategies can be considered as «meta-plans» for dealing with the complexities of frequently occurring design situations. They can be viewed as methodizations or organizations of the primitive design activities identified above. The three types of design strategies considered within the IDEF4 rationale component include:

(1) **External-constraint-driven design**—Design carried out under situations where the goals, intentions, and requirements are not even characterized well much less defined. These situations often result when the designer is brought into the product development process too early.

(2) **Characteristic-driven design**—Design in a closely controlled situation where strict accountability and proof of adequacy are rigidly enforced. These design situations often involve potentially life threatening situations.

(3) **Carry-over-driven design**—Sometimes referred to as «routine» design.

The following classes of knowledge are evident in the practice of system design:

(1) Knowledge of basic principles;

(2) Knowledge of the general design process in a domain;

(3) Knowledge of available components;

(4) Knowledge of previous solution approaches;

(5) Knowledge of available engineering performance models and workable modeling approaches;

(6) Knowledge of test capabilities and results (e.g., what sorts of experimental results and data can be affordably, reliably, or physically acquired);

(7) Knowledge of the human network (i.e., where is the knowledge and information in the organization or in professional associations);

(8) Knowledge of the requirements, design goals, design decision/evaluation process, and design environment of the current problem; and

(9) Knowledge of political or governmental constraints.

In summary, design as a cognitive endeavor has many characteristics in common with other activities such as planning and diagnosis. It can be distinguished by the context in which it is performed, the generic activities involved, the strategies employed, and the types of knowledge

applied. A major distinguishing characteristic is the focus of the design process on the creation (refinement, analysis, etc.) of a *specification* of the end product.

Rationale Diagrams

The Design Rationale Model contains diagrams that describe milestone transformations to design artifacts. Each design situation is represented by a round cornered box with the design state name on the top and a list of diagrams defining the model situation. This strategy is particularly effective as it allows the designer to choose the level of detail for recording design rationale. The diagrams referenced in a design situation box may range from a single diagram illustrating a narrow aspect of the model to all of the diagrams in a design model. If, for example, the designer is making a tactical decision (i.e., a decision that impacts most of the elements in a partition) then the design situation box could contain all diagrams in that partition. The partition may be used in place of the diagrams if the situation contains all of the models in the partition. Typically, the fewer diagrams contained in a design situation, the more detailed the rationale.

For example, consider the following situation in which a designer is captures rationale at a high level of detail. The designer captures the design rationale for the evolution of an *employs/employed by* relation between person class and company class to a link from employee to company. In this case, the «starting» design situation has a reference to the relation structure diagram containing the *employs/employed by* relation, and the «ending» design situation has a reference to the link diagram containing the *employed by* link.

Transitions from one design situation to another are represented by an arc with an arrow pointing in the direction of the transformation. The transition arc lists the observations that necessitated the design change and the changes to be made. For example, in the transition from the design situation containing the *employs/employed by* relation to the situation containing the *employed by* link, the observation will state that the *employs/employed by* relation is not directly implementable and the action will state that the relation will be replaced by the *employed by* link which can be implemented.

Figure 86 illustrates the concepts of design situation, transition and rationale. In this example, an initial design situation (Design State 1) contains the objects *person* and *company* and the *employs/employed by* relation between *person* and *company*. The use cases show that the access rights of employees and non-employees are different. The designer also notes that the relationship between person and company is conditional because not all people are employed, and notes that it would be desirable to define this relationship between employee and company. The designer partitions the object class person into employees and non-employees. This is defined on an inheritance diagram using the subclass/superclass relationship. The designer then redefines the *employs/employed by* relation to be between employee and company and changes its cardinality. These changes result in Design State 2.

In Design State 2, the designer observes that the employs/employed by relation is not directly implementable. There are several options for implementing a relation, including links, arrays, and relation objects. The designer decides to refine the relation to a link. In order to

create the link, the designer embeds referential attributes to each class. The link name is written as L1(R1) which denotes that link L1 was derived from relation R1.

Figure 86
The Observation/Action View of Design Rationale

Design situations are defined on Design Configuration Forms (Table 19). The configuration form allows the designer to name the situation, give a brief description of the design situation, and list all of the diagrams that are important to that situation. The designer can also identify the design situations from which (Entry Rationale) this design situation evolved as well as the one to which it has led.

Table 19. Design Configuration Specification

Author	*J. Smithe*	**Project**	*IICE*	**Date**	*6/1/94*	**Revision**	*3*
Design Situation		Situation Name	*Situation Name*			ID	*ID*
Description	*Description of design situation*						
Diagrams	Entry Rationale		Exit Rationale				
List of diagrams inactive in this in situation	*List of design entry points in terms of Rationale Specifications*		*List of design exit points in terms of Rationale Specifications*				

Rationale forms record information on design transition, such as why a change was needed and which actions are to be taken. Table 20 shows a sample Rationale Specification Form. The form allows the designer to identify the current design situation, name the goal design situation, record a list of observations that prompted the need for a design revision, and state the actions that to be taken. The forms also record the name of the designer, project for which the design is being built, and the date.

Table 20. Rationale Specification

Author	*J. Smithe*	**Project**	*IICE*	**Date**	*6/1/94*	**Revision**	*3*
Rationale		From Design		To Design			
Observation			Action				
List of observations made on design configuration.			List of corresponding actions taken in response to observations made on design configuration				

Rationale Support

Many of the IDEF4 constructs were designed expressly for facilitating deferred decision making. These include (1) the feature taxonomy, (2) relation/link evolution, and (3) the design artifact status designators (®, Δ, and ©). These techniques allow the practitioner to define the artifact with less detail at first and then to gradually evolve towards the more specific detail.

This technique also improves design rationale capture because the design transitions are smoother. The feature taxonomy allows the designer to specify features first in general terms, and then evolve towards specific feature definitions. The design artifact status designators allow the designer to easily bring in objects from the domain and mark them as domain, transition, and completed as they evolve toward a full design specification. Relations allow the designer to specify static structure between objects without committing to implementation. Relations can later be allocated for implementation as links, arrays, or relation objects.

Feature Taxonomy

The feature taxonomy allows features to be characterized in general terms initially, then gradually evolve to a more specific definition as the design evolves. For instance, a designer might first specify a characteristic of a class as a feature. For example, a person class would have a feature called *Name*. As the design evolves, the designer can specialize the definition of *Name* to an attribute and add create, delete, read, and write accessors for *Name*.

Figure 87 illustrates the feature taxonomy using IDEF4's subclass/superclass relation. The features become more specific from left to right. A feature can be an object, event, attribute, link, relation, or method. An object can be a class or an instance, and a class with imbedded objects is called a partition. An attribute can be referential, naming, or descriptive. A referential attribute points to other objects. A naming attribute is an identifier for the object. A descriptive attribute contains values that describe the object's state.

Figure 87
The IDEF4 Feature Taxonomy

IDEF4 DESIGN DEVELOPMENT PROCEDURE

The design process is the predominant activity early in the software system development life cycle. The design artifacts resulting from an IDEF4 design activity are important to both implementation and the subsequent sustaining activities. The following sections discuss the planning and implementation involved in system development and the role the IDEF4 method plays in the design life cycle.

Design Roles and Strategies

Design Roles

According to Tim Ramey's System Development Methodology (SDM) there are two developer roles in system development: technical and administrative (Table 21). Developers in the technical role are concerned with technical excellence. They do not concern themselves with resource constraints, such as time, money, etc. On the other hand, developers in the administrative role are concerned about budget and scheduling constraints. There is a great deal of tension between these role views.

Table 21. Administrative and Technical Roles

Administrative Role	Technical Role
Business Case	Technical Excellence
Return on Investment (ROI)	Technical Merit
Organizational Goals	The Best Product
Standards	Creativity
Life Cycle	Random Life Pattern

The theory of design life cycle helps create an understanding of the basic design processes, particularly for administrative purposes. The design process from such a view is assumed to be cyclical, with steps beginning at some point, continuing through maturity, and eventually ending.

Design Strategies

This view of design as a series of incremental and sequentially interdependent steps is an attempt to order the steps of the process so each step becomes independent, except for its occurrence relative to the other states that surround it. Design strategies can be considered «meta-plans» for dealing with the complexities of frequently occurring design situations (i.e.,

methodizations or organizations of design activities). Three types of design strategies can be considered:

(1) **External-constraint-driven Design** – Design for situations in which the software goals, intentions, and requirements are not well-characterized, much less defined. These situations often result when the designer is brought into the product development process too early.

(2) **Characteristic-driven Design** – Software design in a closely controlled situation for which strict accountability and proof of adequacy are rigidly enforced. These design situations often involve potentially life-threatening situations.

(3) **Carry-over-driven Design (Routine Design)** – Changes to existing, implemented designs or designs that are well understood (e.g., sorting).

From an OOD point of view, the external-constraint-driven and carry-over-driven strategies are the most common design situations. The design development procedure, outlined in the following sections, is a distillation of the experience and insights gained in building several IDEF4 designs on case studies representative of situations requiring these design strategies.

Strategizing the Developer Focus

The first step in a design strategy is deciding on the developer focus. The type of design and the stage the design is in dictates how much emphasis is put on the designer as an administrator or technical developer. The focus of the design on these roles should be decided and implemented in the strategy to manage expectations and prevent differing goals for the design.

From the administrative perspective, development is a business venture. It is a very structured undertaking that takes place over a fixed period of time and within fixed resource limitations. The system development process is characterized by line items and milestones. The administrative developer deals with contractual requirements (schedules, budgets, delivery items) and company goals and guidelines. Tradeoffs are made between the constraints of costs and schedules and technical excellence.

The technical developer, on the other hand, works to discover the problem, develop a solution, create a product, and implement a system. Constraints are viewed in terms of development tools, not time and cost. A technical developer may revisit the discovery of the problem and solution over and over without ever moving on to the creation of an actual product (Figure 88).

Figure 88
Developer Focus at the Design Level

A design strategy should include administrative and technical viewpoints. If the software is consumer-oriented, the administrative development characteristics are more applicable and should be weighted heavier than the technical development characteristics. In a research and development environment, the opposite is true. In the initial stages of design, the technical development view also should be weighted more heavily. As the design solidifies and the product becomes tangible, the focus should switch to the administrative characteristics.

IDEF4 Design Evolution Process

The development of an IDEF4 design is a process of specifying the structure and behavior of an object-oriented program or system. The process involves the use of design layers containing partitions which help manage the complexity of the design. The design layers address the administrative role of a developer. A design layer has five design activities that evolve the design from the initial stages to implementation. These five design activities address the technical role of the developer.

Organization of IDEF4 Design Layers

The IDEF4 method uses a multi-dimensional approach to object-oriented software system design (Figure 89). These dimensions are: (1) design layers (requirements analysis, system, application, and low-level design), (2) artifact design status (application domain, in transition, software domain), and (3) design models (static models, dynamic models, behavior models, and rationale component)[18]. The design models may be developed or they may be re-used from other projects. The method has explicit syntactic support for these dimensions. The design procedure is organized around the layers or levels of design.

[18]Setting criteria for completion of the design is important in the design process. This is particularly true of modeling situations to prevent never-ending model development. It is not possible to give precise criteria for the completion of design activities, but as the design approaches completion, the rate of change of the design will decrease. This occurs because each new design constraint that is specified places more and more restrictive requirements on the design. When this occurs, the model development should taper off.

Figure 89
IDEF4 Design Layers Relative to Model and Status

Each layer (System, Application, and Low-Level) contains a Static Model, Dynamic Model, Behavior Model, and Design Rationale. These layers can be viewed as steps in the design process because they track the design from the highest level of abstraction (System-Level) to the actual software components (Low-Level). The role of the developer will vary during the different phases of design based on the type of design being created. Focus on an administrative or technical view will also affect the products resulting from the design activities.

Design Activities

In general, when using IDEF4 to develop an object-oriented design, the following five general design activities[19] are applied iteratively[20] to each layer:

[19]The sequence of the design activities is not implied by the order in which they appear.

[20]The term «iterative application» implies that the same process of specification is normally applied to each element in the partitioning of the five design types as well as to the overall design development activity. This iteration continues until the prototype classifications of the resulting elements can be clearly established.

(1) Partition – Partition the evolving design into smaller, more manageable pieces by like behavior. This is where objects are identified.

(2) Classify/Specify – Classify the design against existing definitions or specify a new design object. This is where object behavior (methods, state) is defined or where objects are compared and contrasted against existing objects.

(3) Assemble – Incorporate design objects into existing structures. This is where relations, links, and dynamic interactions between objects are identified.

(4) Simulate – Validate and verify the models by exercising them against use scenarios in the requirements model. This is where problems with the current design are identified and described.

(5) Rearrange – Rearrange existing structures to take advantage of newly introduced design objects. This may result in an adjustment in the design partitioning which would cause another iteration in the design recursion.

Partition

IDEF4, as an OOD method, focuses on the identification of the objects and the behavior required of a system. The focus on behavior to be exhibited by the objects provides an appropriate means of partitioning systems into small, easily understood pieces. When objects are partitioned by behavior, a more modular design is provided for; this results in implementations that have the desirable life cycle characteristics for which object-oriented implementations are known.

The identification of objects starts by focusing on the problem from the application domain and looking for the *things* in the problem that behave similarly. These things are likely to fall into five categories (Schlaer, 1988) :

(1) physical or tangible objects,

(2) roles or perspectives,

(3) events and incidents,

(4) interactions, and

(5) specifications and procedures.

Objects may be partitioned several times; for example the instances of person can be partitioned into employed or unemployed, male or female, or resident or alien. The way the objects are partitioned depends on the purpose of the system to be built.

Classify/Specify

In the classification/specification activity, behaviors and objects identified through partitioning are matched against existing models to classify them. If an object cannot be classified, a specification must be designed. For example, if the objects employed and unemployed have been identified, then they should be classified against the class person. Thus the specifications for employed and unemployed will be of the form: «employed is the same as person except . . .» or «unemployed is the same as person.»

Model specification will become more and more complete as this generic activity is revisited throughout the design cycle. Specifications in each cycle may be viewed as additional requirements in the next design cycle.

Assemble

Design artifacts that have been classified or specified must be assembled into IDEF4 organizational structures. Newly classified/specified objects must be assembled into new or existing models. When designing large systems, assembly activity may occur in joint meetings between individuals responsible for different aspects of the design. If unemployed and employed are «the same as person except . . .», then they should appear as subclasses in the inheritance diagram containing person.

Simulate

Design artifacts that have been assembled into models must be run against use scenarios to reveal possible design flaws. In IDEF4, the use cases are contained in a requirements model which is built using IDEFØ and IDEF3. Simulation validates and verifies current design model. This activity should primarily uncover flaws in the Dynamic and Behavior models of IDEF4. Flaws discovered in these models may, however, motivate changes to the structure of the Static model. The reason for this is that the object model stabilizes fairly quickly as the design evolves.

Rearrange

The rearrange activity is similar to applying annealing to the models. Introduction of new model elements over time causes «brittleness» to build up in the models, so that it becomes increasingly difficult to extend the models. Rearrangement simplifies the models and makes them more flexible.

In the rearrange activity, designers look for similarities in artifact structure to uncover opportunities for reuse. Rearrangement may necessitate alterations to the specifications of classes, methods, and their associated organizing structure. Designers may, for example, combine behavior models and examine the specifications associated with each to identify possible refinements. These refinements generally involve moving constraints up and down the taxonomies while being careful to ensure there are no changes to the behavior of the superstructure or the substructure and no conflicts with any evolving system requirements.

The five design activities described above will be applied iteratively to each design layer. The design layers are viewed as design phases which move the project from the initial stages to the final product while simplifying the management of the design.

IDEF4 Phases of Design

IDEF4 has four phases to reduce the complexity of the system design:

- **Phase 0** – Analysis Requirements
- **Phase 1** – System Design Layer
- **Phase 2** – Application Layer
- **Phase 3** – Low-Level Design Layer

An IDEF4 design starts with the analysis of requirements (Phase 0) which results in a requirements model[21]. The requirements model consists of a function model (IDEFØ) used for scoping, and use scenarios (IDEF3)[22]. Elements in these models are mapped back to requirements so that they can be used to validate the models. Domain objects identified during the requirements analysis phase are input to the IDEF4 design. They are encoded in IDEF4 form and marked as domain. As computational objects are developed, they are marked as transitional, and as the design solidifies, they become completed objects. The level of completion of an IDEF4 design is determined by the status of individual artifacts in the design (i.e., as the objects in the design stabilize, the design itself stabilizes).

The other three distinct design phases are (1) the system layer, (2) the application layer, and (3) the low-level layer. These design layers reduce the complexity of the design. The system layer (Phase 1) ensures connectivity to other systems in the design context. The application layer (Phase 2) models the interfaces between the components of the system. These components include commercial applications, previously designed and implemented applications, and applications to be designed. The low-level design layer (Phase 3) is responsible for the foundation objects of the system.

In each of these phases, IDEF4 follows an iterative procedure in which the design is partitioned into objects and the external specification for each object is developed so that the internal specification of the object may be done concurrently. After partitioning, the objects are assembled; that is, static, dynamic, and behavioral models detailing different aspects of the interaction between objects are developed. After the models are developed, it is possible to simulate interaction scenarios between objects to uncover design flaws. The existing models are then rearranged and simulated until the designer is satisfied.

---

[21]The requirements model is explained in more detail in Chapter 7.

[22]For a full description of the IDEFØ and IDEF3 methods, readers should refer to (Mayer, 1990) and (Mayer, 1992).

Phase 0 Analyze System Requirements

The responsibility of this phase is to develop the system requirements and domain models into a set of objects, relations, and use scenarios that can be used as a starting point for design. These objects will be included in the system, application, and low-level phases of design as starting objects. An important part of analyzing requirements is building use scenarios, also known as use cases. Use scenarios are models of the system users' processes. The set of use scenarios should cover and the functional requirements of the system in order to provide a set of test scenarios for validating the design and verifying the implementation. If the design or implementation fails to adequately support a use case, then the associated requirements may easily be identified.

Phase 1 System-Level Design

The system-level design starts once the domain (raw material) objects have been collected. This develops the design context, ensures connectivity to the legacy system, and identifies the applications that must to be built to satisfy the requirements. Static models, dynamic models, behavioral models, and the design rationale are built for the objects at the system level. These specifications become the requirements on the next level of design, namely the application-level design.

Phase 2 Application-Level Design

The application-level design identifies and specifies all the software components (partitions) needed in the design. Static models, dynamic models, behavioral models, and the design rationale are built for the objects at the application level. These specifications become the requirements on the next level of design, namely the low-level design. This level lays out the top-level objects of the software being designed. These top-level objects can include mid-level commercial components (e.g., a communications program, an image viewer, or a fax modem program).

Phase 3 Low-Level Design

Static models, dynamic models, behavior models, and the design rationale components are built for the objects at the low-level design. Sublayers may be built within each layer to reduce complexity. The low-level design may use low-level commercial technology (e.g., buttons, sliders, windows, forms, and input fields). It is responsible for the foundation classes of the design. A great deal of design object reuse should occur at this level because the designers use foundation objects from other designs.

Using Other IDEF Methods in Analysis

Clearly stated system requirements are a valuable source of assistance to the analyst in understanding customers' expectations of the system. However, because the customer can not always be expected to have full knowledge of the true nature of the problem involved, designers should not limit themselves to the requirements document alone. Quick and accurate

identification of the initial classes and methods is critical to both the successful completion of the design and a full understanding of the nature of the problem; therefore, it is often advantageous to look to other sources of information on the proposed system and its environment. These sources may include function, process, and information models as well as existing object-oriented designs (OODs). The IDEF family includes methods for constructing each of these types of models.

For function modeling, IDEFØ can be used to assist designers in identifying concepts and activities that should be addressed by the system. IDEF1X can be used by designers to develop an understanding of the information the organization uses. IDEF3 provides for process flow and object state descriptions that will assist designers to organize the concepts of the existing system, the proposed system, user interaction with the system, and the state changes objects undergo in the system.

Applying IDEFØ (Function Model) to Object-Oriented Design

IDEFØ is a method for capturing a static view of the functions performed by an organization. It also captures classification mechanisms for describing the roles objects play relative to those functions. IDEFØ supports analysis of an organization or system from a functional perspective. As an analysis method, IDEFØ can be particularly useful in domain analysis, an activity which precedes design. It is not particularly well-suited for object-oriented design, however. Use of IDEFØ in the role of an object-oriented design method should be avoided. Although the functions that the projected system must perform are important, too much emphasis on the system functions can produce system designs that are functionally-oriented rather than object-oriented. A system organized around functional decomposition tends to consist of tightly coupled modules that are difficult and costly to maintain, thus eliminating a major advantage of the object-oriented paradigm. Minor changes can mean major system rewrites.

Although IDEFØ should not be seen as a vital component to the development of object-oriented systems, the information captured in a function model can aid the designer in the early stages of the project, particularly during requirements analysis. IDEFØ can provide valuable insights into initial classes and routines that the system requires. It can also assist the designer in scoping the system to see what should be included in the project. This will give the project a more tangible start and finish point so that the system transitions from the development phase to actual implementation. IDEFØ helps the designer understand the business of the end user, giving the designer information on the business the system will be supporting. This is helpful when making decisions on execution.

Since IDEFØ modeling generally involves extensive use of function decomposition, it is advisable to limit the depth of decomposition to two or three levels. Functions from the model may represent methods. The concepts should be examined as classes or object instances (Figure 90). Reading the definitions and text included with the model may also give the designer some insight into what information will be needed in the system.

In the diagram shown in Figure 90, the Create, Reference, Update, and Delete activities may correspond to Create, Reference, Update, and Delete methods in the OOD. If this particular diagram is decomposed any further, then the decomposition would be by type. It is possible that the Object concept may consist of a bundle of different kinds of objects which would be useful for defining the inheritance lattice in an OOD. The Authorization concept with subtypes Read Authorization and Write Authorization could also be used in the OOD inheritance lattice. The Object input to Update and the Updated Object output is indicative of a state change and could be used in an OOD state model. The concepts Pointer, Object, User, Command, and Authorization could also be objects in the resulting OOD. Figure 90 illustrates how useful an IDEFØ model could be in an OOD, but these models are most useful when used to validate designs and verify implementations, because they define what activities the system must support.

Figure 90
Diagram Based on IDEFØ

Applying IDEF1X (Data Model) to Object-Oriented Design

The IDEF1X method for data modeling provides input in the area of class identification and links between classes. By re-interpreting the semantics of entity to mean class, relation to mean link, categorization to mean inheritance, foreign key to mean referential attribute, key attribute to mean naming attribute, it is not difficult to directly transliterate IDEF1X models to static models in IDEF4. This could be useful for organizations that have large inventories of IDEF1X models.

However, the IDEF1X method was developed for designing relational database schemas. Herein lies the problem: IDEF1X models exhibit a commitment to a certain style of implementation, so models translated from IDEF1X to IDEF4 need to be reverse engineered to a certain extent to uncover the object model. The entities in IDEF1X are identified on the basis of descriptive attributes, such as state; in IDEF4, objects are identified on the basis of their behavior, leaving the attributes as an implementation decision on how to represent the object's state. IDEF1X associative entities are identified to resolve many-to-many relations, so from an OOD perspective they may appear contrived. IDEF1X relations represent an implementation choice; that is why IDEF1X relations are equivalent to IDEF4 links. IDEF4 links are an implementation decision based on IDEF4 relations, thus IDEF1X relations must be reverse engineered to find the IDEF4 relation underlying the link. Figure 91 illustrates this point by showing the IDEF1X *Employs/Employed By* relation with its equivalent IDEF4 link. Note the referential attribute is playing the same role as the foreign key in the Employee class. Because the IDEF1X relation is one of many possible implementations, it is important to state the IDEF4 relation on which the IDEF4 link could be based. The IDEF4 relation is important in that it will allow a common take-off point for investigating other implementations.

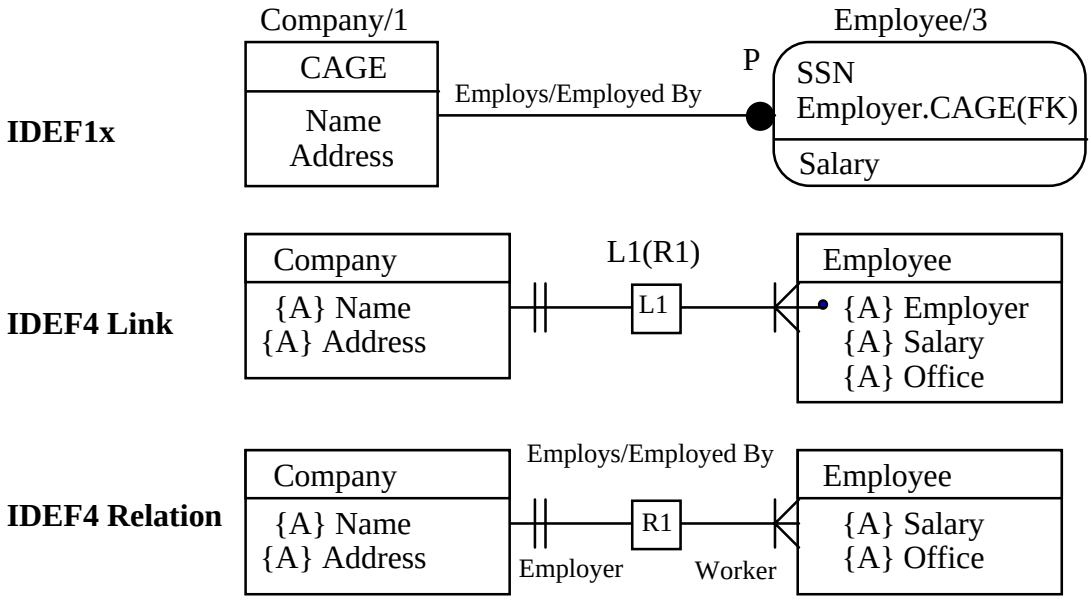


Figure 91
IDEF4 Equivalent Representations to IDEF1X Relation

IDEF1X models translate to IDEF4 static models; there is no information on the object dynamics. In IDEF1X the foreign key is imbedded in the dependent entity because relational databases rely on queries for joining data across tables. In OOD the referential attributes are placed on the independent and/or independent class as object-oriented systems rely on navigation across the network of links between objects.

The categorization relationships in IDEF1X can give the designer an idea of possible subclass/superclass relationships. The categorization relationship also depicts which attributes can be stored by the superclass and do not need to be included in the subclass. Other relationships depicted in IDEF1X can be used as well. Figure 92 shows the equivalent IDEF4 syntax for an IDEF1X categorization.

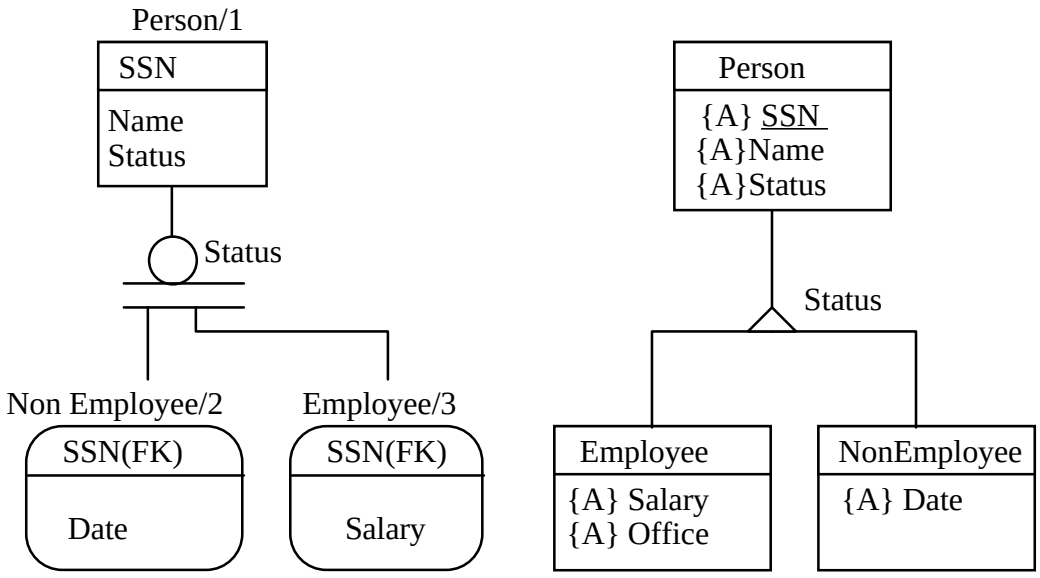


Figure 92
IDEF1X Categorization and Equivalent IDEF4 Inheritance

Different views created in IDEF1X may give the designer an idea of where partitions can be used around objects. Definitions for the entities and attributes in an IDEF1X model may contain information about objects, classes, methods, and events. An IDEF1X model will give the designer information pertinent to the static model. However, as with the IDEFØ method, the modeler should use his/her best judgment when doing this because the semantics of the original model elements are not exactly the same as IDEF4.

Applying IDEF3 (Process Model) to Object-Oriented Design

Perhaps the most useful IDEF method for the object-oriented designer is IDEF3. IDEF3 provides the designer with a method for developing requirements models containing use scenarios. Each use scenario contains process flow descriptions detailing the required or intended use of a particular design partition. Because IDEF3 models describe how objects participate in processes, it provides useful information for developing the Client/Server models.

Interaction between the user and the system must be considered in design to ensure that the system meets user requirements. Object-oriented system designers must consider the man-machine interface. Modern graphical user interfaces are event driven, so user processes are not explicitly enforced. This style of interface is very flexible as it allows many different work-flow scenarios.

Another useful element of the IDEF3 method is the object state transition network (OSTN) diagram. The OSTN helps identify the behavior that will exist in the system. It also assists the designer in determining pre- and post-conditions for methods, design of transactions, and state changes in the system objects.

REFERENCE LIST

Booch, G., (1991). Object-Oriented Design With Applications. Redwood City, CA: The Benjamin/Cummings Publishing Company.

Coad, P. and Yourdon, E. (1991) Object-Oriented Design. Englewood Cliffs, NJ: Yourdon Press.

Coleman, D., et al. (1994) Object-Oriented Development: The Fusion Method. Englewood Cliffs, NJ: Prentice Hall.

Jacobsen, I. (1994) Object-Oriented Software Engineering: A Use Case Driven Approach. Reading, MA: Addison-Wesley.

Keen, A.A., Mayer, R.J., Approaches to design object management, Society of Manufacturing Engineers, AutoFact '91, Chicago, IL, November 10 - 14, 1991.

Knowledge Based Systems Laboratory. (1991). IDEF4 technical report (KBSL-89-1004). College Station, TX: Department of Industrial Engineering, Texas A&M University.

Korson, T. D. and Vaishnava, V. K. (1986). An empirical study of the effects of modularity on program modifiability. E. Soloway & S. Iyengar, (Eds.), Empirical studies of programmers. Ablex Publishers.

Martin, J. and Odell, J. (1992) Object-Oriented Analysis and Design. Englewood Cliffs, NJ: Prentice Hall.

Mayer, R. J., et al. (1987). Knowledge-based integrated information systems development methodologies plan (Vol. 2) (DTIC-A195851).

Mayer, R. J. (1991). Framework foundations research report [Final Report]. Wright-Patterson Air Force Base, OH: AFHRL/LRA

Mayer, R. J., Menzel, C. P., and deWitte, P. S D. (1991). IDEF3 technical report. WPAFB, OH: AL/HRGA.

Meyer, B. (1987). «Reusability: The case for object -oriented design.» IEEE Software, 4(2), 50-64.

Rumbaugh, J. (1991) Object-Oriented Modeling and Design. Englewood Cliffs, NJ: Prentice Hall.

Shlaer, S. and Mellor, S. (1992) Object Lifecycles: Modeling The World in States. Englewood Cliffs, NJ: Yourdon Press.

Shlaer, S. and Mellor, S. (1988) Object-Oriented Systems Analysis: Modeling The Real World in Data. Englewood Cliffs, NJ: Prentice Hall.

Taylor, D., (1990) Object-Oriented Technology: A Manager's Guide. Reading, MA: Addison-Wesley.

Udell, J. (1994) Componentware. Byte, May 1994, 46-56.

Yourdon, E., and Constantine, L. L. (1979). Structured design: Fundamentals of a discipline of computer program and systems design. Englewood Cliffs, NJ: Prentice-Hall.

Zachman, J. (1987). A framework for information systems architecture, IBM Systems Journal, 26(3), 276-292.

ACRONYMS

CLOS	Common Lisp Object System
COTS	Commercial Off-the-Shelf Technology
DDE	Dynamic Data Exchange
DoD	Department of Defense
DM	Dynamic Model
DRM	Design Rationale Model
DTIC	Defense Technical Information Center
ExSpec	External Specification
FORTRAN	**For**mula **Tran**slator
ICAM	Integrated Computer-Aided Manufacturing
ID	Identifier
IDEF	Integrated Definition
IICE	Information Integration for Concurrent Engineering
InSpec	Internal Specification
IISEE	Integrated Information Systems Evolutionary Environment
ISO	International Organization for Standardization
IT	Information Technology
KBSI	Knowledge Based Systems, Inc.
LDFD	Logical Data Flow Diagramming
LISP	**Lis**t **P**rocessing
MIT	Massachusetts Institute of Technology
BM	Behavior Model
ODBC	Open Database Connectivity

ODBMS	Object DataBase Management System
OLE	Object Linking and Embedding
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OOPL	Object-Oriented Programming Language
OSTN	Object-State Transition Network
ROI	Return on Investment
SM	Static Model
TQM	Total Quality Management
UOB	Unit of Behavior

TRADEMARK NOTICE

DDE™ is a trademark of MicroSoft, Inc.

OLE™ is a trademark of MicroSoft, Inc.

ODBC™ is a trademark of MicroSoft, Inc.

APPENDIX A: LINEAR SYNTAX

The linear syntax of IDEF4 has two parts: 1) the syntax used for labeling design artifacts on diagrams, and 2) the syntax for specifying design artifacts. The label syntax describes the valid form of a class label and attributes in a class box on a diagram, whereas the artifact syntax defines a valid form for describing the class artifact.

IDEF4 Label Syntax

The label syntax provides the syntax for labeling design artifacts on IDEF4 diagrams and for referencing design artifacts from diagrams.

Names

Artifact_name =>	String
Partition_name =>	Artifact_name
Class_name =>	Artifact_name
Instance_name =>	Artifact_name
Variable_ID =>	'?' *Variable_Name*
Instance_ID =>	{*Instance_Name* | Variable_ID}
Design_Status =>	{® | Δ | ©}

Paths

Partition_Path =>	[{'>' | '<'+}][*Partition_Name* '>']*
Simple_Partition =>	[Partition_Path] *Partition_Name*
Aliased_Partition =>	Simple_Partition
Simple_Class =>	[Partition_Path] *Class_Name*
Aliased_Class =>	Simple_Class

Design entities

Partition =>	Simple_Partition ['(' Aliased_Partition ')']
Class =>	Simple_Class ['(' Aliased_Class ')']
Instance =>	Class '[' Instance_ID ']'

Instance_State =>	Instance *State_Label*
Attribute =>	[{Class | Partition | Instance} '.'] *Attribute_Name*
Behavior =>	[Partition ':'] *Message_Name*
Method =>	[{Class | Partition} '.'] *Message_Name* ['[' *Method_Name* ']']

Types

Basic_Type =>	{*integer* | *real* | *string* | *boolean* | ....}
Attribute_Type =>	{Basic_Type | Class}
Parameter_Type =>	Attribute_Type

Entity Labels

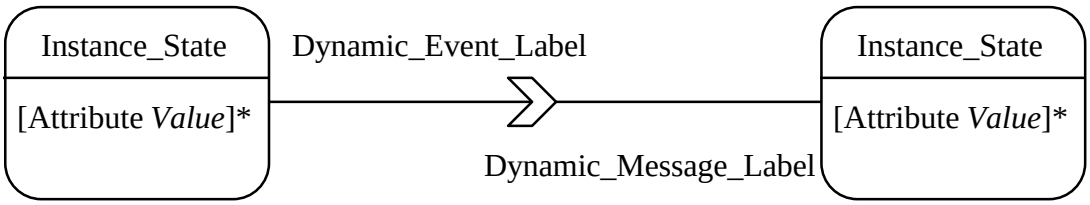
Partition_Entity =>	Partition [Design_Status]
Class_Entity =>	Class [Design_Status]
Instance_Entity =>	Instance [Design_Status]
Instance_State_Entity =>	Instance_State [Design_Status]
Attribute_Entity =>	Attribute [Design_Status]
Behavior_Entity =>	Behavior [Design_Status]
Method_Entity =>	Method [Design_Status]

Dynamic Model Diagram Labels

Dynamic_Event =>	['E' *Event_Number* ':'] [{Partition | Class} '.'] *Event_Name*
Dynamic_Message =>	['M' *Message_Number* ':'] Method
Dynamic_Event_Label =>	Dynamic_Event [Design_Status]
Dynamic_Message_Label =>	Dynamic_Message [Design_Status]

Figure 93
Client/Server Diagram Syntax

Instance State Diagram Syntax



Example

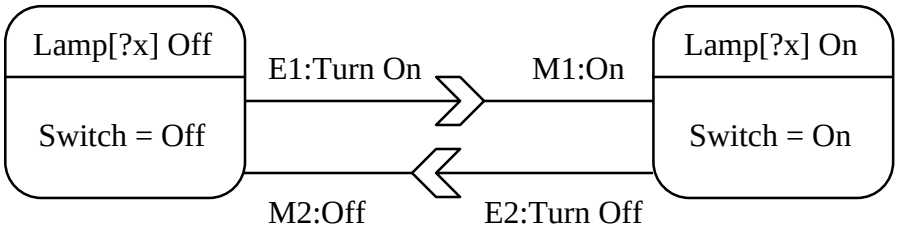


Figure 94
State Diagram Syntax

Static Model Diagram Labels

Supporting Syntax

Relation_List =>	'(' Static_Relation [',' Static_Relation]* ')'
Static_Relation =>	['R' *Relation_Number* ':'] [Partition '.'] [*Relation_Name*] [Relation_List]
Static_Link =>	['L' *Link_Number* ':'] [Partition '.'] [*Link_Name*] [Relation_List]
Return_Parameter =>	*Parameter_Name* [':' Feature_Type]
Specialized_Class_Parameter =>	{'(' *Parameter_Name* Class ')' | Class}
Class_Parameter_List =>	':' Specialized_Class_Parameter Specialized_Class_Parameter+
Static_Feature =>	«{}» *Feature_Name*
Static_Object =>	«{O}» *Object_Name*
Static_Attribute =>	«{A}» *Attribute_Name* [':' Attribute_Type]
Static_Single_Method =>	«{M}» *Message_Name* ['[' *Method_Name* ']']
	['\n' Return_Parameter]*
Static_Multi_Method =>	«{MM}» *Message_Name*
	['\n' '[' *Method_Name* [Class_Parameter_List] ']']
	['\n' Return_Parameter]*
Static_Class =>	«{C}»*Class_Name* ['(' Aliased_Class ')']
Static_Instance =>	«{I}» Instance
Static_Partition =>	«{P}» *Partition_Name* ['(' Aliased_Partition ')']
Static_Event =>	«{E}» ['E' *Event_Number* ':'] *Event_Name*

Diagram Labels

Static_Relation_Label =>	Static_Relation [Design_Status]
Static_Link_Label =>	Static_Link [Design_Status]
Static_Feature_Label =>	Static_Feature [Design_Status]
Static_Attribute_Label =>	Static_Attribute [Design_Status]
Static_Method_Label =>	{Static_Single_Method | Static_Multi_Method} [Design_Status]
Static_Class_Label =>	Static_Class [Design_Status]

Static_Instance_Label => Static_Instance [Design_Status]

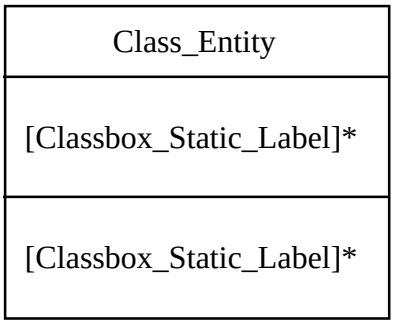
Static_Partition_Label => Static_Partition [Design_Status]

Static_Object_Label => Static_Object [Design_Status]

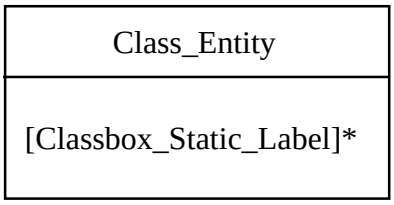
Static_Event_Label => Static_Event [Design_Status]

Classbox_Static_Label => {Static_Feature_Label | Static_Attribute_Label |
Static_Method_Label | Static_Class_Label |
Static_Instance_Label | Static_Partition_Label |
Static_Object_Label |Static_Event_Label}

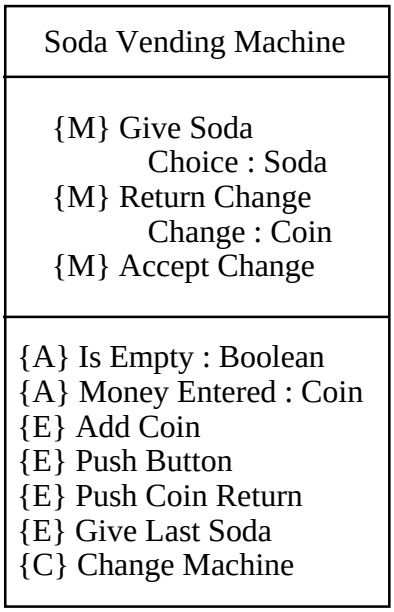
Static Class Box Syntax



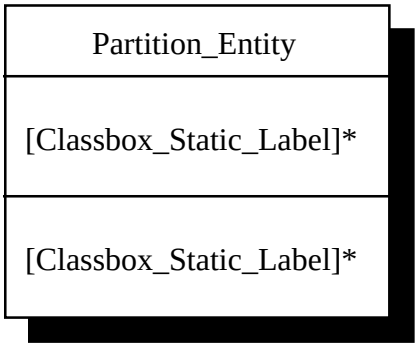
or (everything is public)



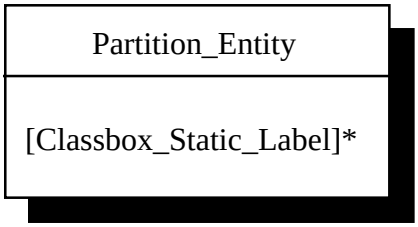
Example



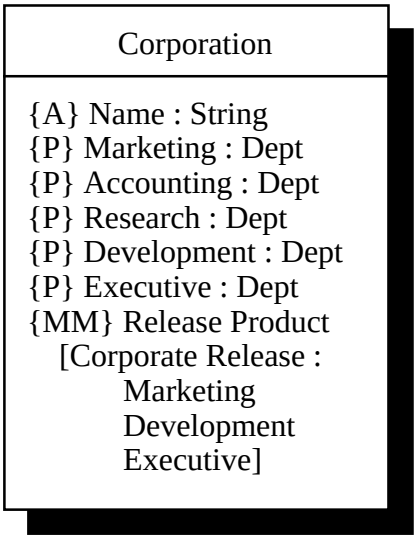
Static Partition Box Syntax



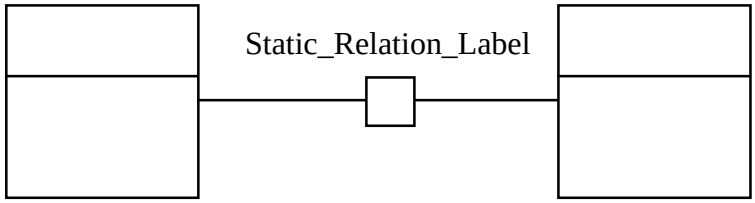
or (everything is public)



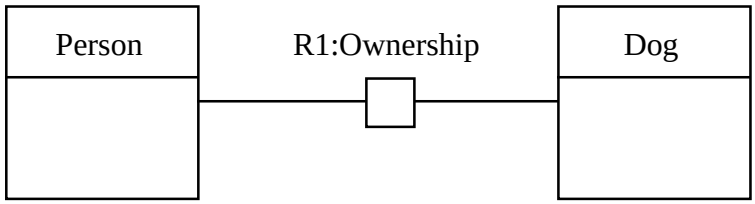
Example



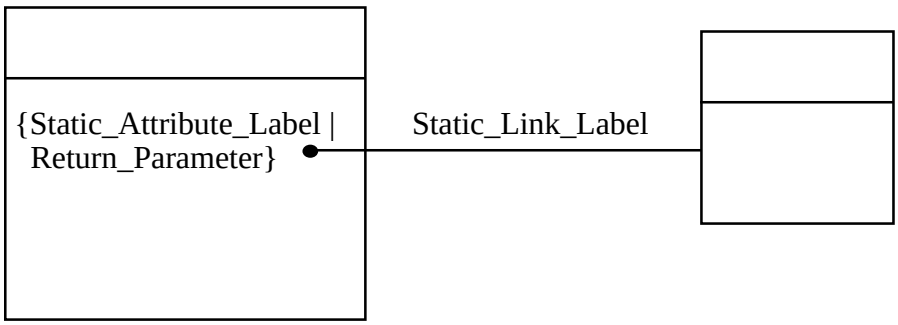
Static Relation Syntax



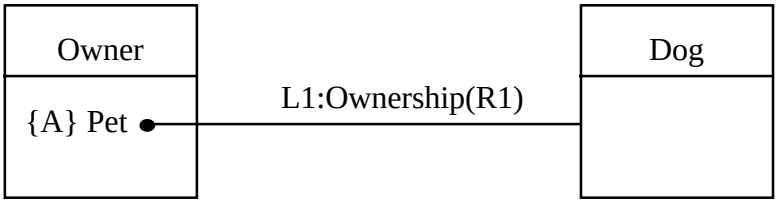
Example



Static Link Syntax



Example

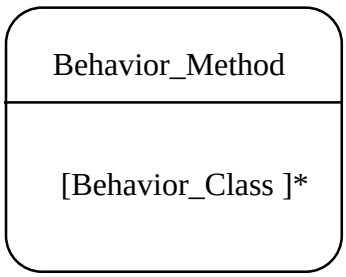


Behavior Model Diagram Labels

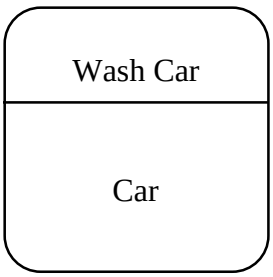
Behavior_Method => [Partition_Path] *Method_Name* [Design_Status]

Behavior_Class => Class [Design_Status]

Method Box Syntax



Example



Artifact Specification Syntax

Method Specification

{

Name: *Method_Name*

Behavior: Behavior

Author: *Author*

Date: *Date*

Specialization Classes

{

[Class ‘:’ *Variable_Name*]*

}

Input Parameters

{

[Parameter_Type ‘:’ *Variable_Name*]*

}

Output Parameters

{

[Parameter_Type ‘:’ *Variable_Name*]*

}

Parent Methods

{

[*Method_Name* [‘:’ Class*]]*

}

Child Methods

{

[*Method_Name* [‘:’ Class*]]*

}

Definition/Description

{

[*Textual_Description*]

}

External Specification

{

[*External_Constraint*]*

}

Internal Specification

{

[*Internal_Constraint*]*

}

}

Class Specification

{

Name: *Class_Name*

Partition: Partition_name

Author: *Author*

Date: *Date*

Super Classes

{

[Class]*

}

ClassFeatures

{

ClassAttributes{[Attribute Specification]}

ClassMethods{[Method Specification]}

}

InstanceFeatures

{

Attributes {[Attribute Specification]}

Methods{[Method Specification]}

Objects{}

Instances{}

Classes{}

Partitions{}

Events{}

}

PartitionFeatures

{

MultiMethods{}

Relations{}

Links{}

}

Definition/Description

{

[*Textual_Description*]

}

External Specification

{

[*External_Constraint*]*

}

Internal Specification

{

[*Internal_Constraint*]*

}

}

APPENDIX B: IDEF4 GLOSSARY

Application Domain	The domain the application is being designed for (i.e., the end users).
Attribute	An implementation choice on how to represent an object's state.
Behavior Diagram	Defines relations between similar behavior.
Behavior Model	Defines the relations between the similar behaviors of objects.
Behavior Specification	Specification for the signature and internal implementation of a method.
Class	A description of a group of objects with common behavior, relationships, and semantics.
Client/Server Relationship	The dynamic relationship between an object that sends a message (client) and an object that receives it (server) and acts on it.
Design Artifact	Items used in the design.
Design Domain	The domain in which the design is performed.
Design Method	A heuristic guide to thinking that applies a certain philosophy of design (Design Methodology).
Design Methodology	A philosophy of design (see also Design Method).
Design Rationale Component	Provides a top-down representation of the system, giving a broad view that encompasses the three design models and documents the rationale for major design evolutions.
Domain	Scope of the system being developed.
Dynamic Model	Specifies the communication between objects and the state transitions of objects.
Encapsulation	The act of imbedding features in objects.

Event	A signal generated by a method in an object indicating some condition in the object.
Feature	A catch-all category for concepts that the engineer wants to capture in the software design.
Function	An operation that returns a value and has no side effects.
Implementation Domain	The environment and language in which the design is implemented and the environment in which the implementation executes.
Inheritance	An object-oriented mechanism for sharing attributes and methods between objects.
Inheritance Diagram	Illustrates the subclass/superclass relation between classes.
Instance	An object described by a class.
Link	The association formed by a referential attribute imbedded in one class whose domain is another class.
Link Diagram	A diagram depicting links between objects.
Link Instance	The association formed by a referential attribute of one instance pointing to another instance.
Message	A token sent from one object to another to requesting an action.
Method	An implementation of behavior, such as a set of instructions for the object to perform some operation, or a procedure contained in an object.
Object Class	A description of a group of objects with common behavior, relationships, and semantics.
Object Instance	An object described by a class.
Object Life Cycles	The pattern of behavior that an object exhibits from creation to deletion.
Partitions	A *partition object* contains objects and relations.

Polymorphism	The property of an operation to behave differently on different objects.
Procedure	An operation that returns no value and has side effects.
Relation	An association between objects.
Relation Diagram	A diagram depicting relations between objects.
State	The values of the attributes of an object at a particular time.
State Diagram	A diagram depicting transitions between states.
Static Model	Defines time-invariant relations between objects, such as ownership and inheritance. The static model is described using inheritance, relation, and link diagrams.
SubClass	Grouping particular instances of a class into an even more specialized class.
SuperClass	The class from which a subclass is derived.