DOMAIN-SPECIFIC PROGRAMMING: AN OBJECT-ORIENTED AND KNOWLEDGE-BASED APPROACH TO SPECIFICATION AND GENERATION

Neil Allen Iscoe

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-90-37 December 1990

DOMAIN-SPECIFIC PROGRAMMING: AN OBJECT-ORIENTED AND KNOWLEDGE-BASED APPROACH TO SPECIFICATION AND GENERATION

Neil Allen Iscoe, Ph.D.

The University of Texas at Austin, 1990

Supervising Professor: J. C. Browne

Programmers must have an understanding of both programming knowledge and application domain knowledge to write application programs. But while programming is well enough understood to model and teach, application domain knowledge is not yet well understood, and is codified only in an informal ad hoc manner. Because representations that precisely characterize application domain knowledge do not currently exist, errors are frequently made when gathering and mapping specifications from the informal to the formal.

This dissertation defines a meta-model for application domain knowledge and describes a methodology for its instantiation into domain-specific models. Domain models are representations of application domains that can be used for a variety of operational goals in support of specific software engineering tasks or processes. The meta-model and methodology in this dissertation facilitate understanding and analyzing application areas and eliciting and formalizing software requirements and specifications. The emphasis is on general characterization techniques that can be used to instantiate models from different application domains.

The meta-model begins with primitive *attributes* that capture the semantics of domain properties in terms of scales (from mathematical measurement theory); units, quantities, and granularity (from the physical sciences); population parameters (from statistics); and value set transitions. The next level constructs are *classes* that encapsulate a set of attributes, provide for the definition of derived attributes, allow additional operations to be defined, are responsible for object instantiation and deletion, provide for the definition of axioms, and maintain summary statistics on object sets.

Domain models are instantiated by representing specific application domains in terms of attributes and classes. Classes are organized and structured using hierarchical decomposition to create subclasses, and composition and association to construct larger classes from smaller ones. Examples are given in terms of library, accounts receivable, and other related problems.

Acknowledgements

This dissertation would not have been possible without the assistance and guidance of J.C. Browne, who by knowing what he sees has helped me to do the same. I also thank John Werth, who helped me to translate informal ideas into formal definitions without losing perspective. I can no longer even guess at the number of weekend meetings which were an integral part of this dissertation research. I can never thank these two professors enough.

I would also like to extend thanks to my other committee members for their valuable comments on the dissertation proposal, drafts, and final defense. In particular, I would like to thank David Barstow for his comments on domain modeling, multiple inheritance and part-of relationships, Woodrow Bledsoe for his comments on knowledge representation issues, and Al Dale for his comments on the accounts receivable example.

Although not a member of my committee, Gerry Williams played an integral part in this dissertation. He provided an independent verification and validation of the accounts receivable model, as well as providing insightful comments on several other aspects of the dissertation.

Bill Curtis has been both a mentor and a friend. I am grateful for his guidance, support, enthusiasm, and technical contributions to all areas of my research.

At the risk of leaving someone out, I'd also like to list those others that reviewed sections of the dissertation, provided encouragement, or other valuable insights. These people include in alphabetical order: Bob Balzer, Susan Gerhart, John Hartman, Mitch Lubars, Jean Ludl, Bob McKenzie, Colin Potts, Charlie Richter, Bill Robinson, Richard Waters, and Larry Vansickle.

As indicated by the dedication, this dissertation would not have been possible without my parents and grandmother, whose encouragement helped me to finish this and many other projects. I'd also like to thank my sister Ellen, and my brother Craig for their continual support in this and other areas of my life.

Finally, I would like to thank those friends who provided encouragement, typing and drawing assistance, and moral support: Rick Alterman, Nader Bagerzadeh, David and Cheryl Conley, Jay Gribble, Gina McVay, Jonathan Mack, Nancy MacMahon, Ida Miller, Jim Middleton, Lori Najvar, Jayne Noble, Lila Anne Parker, Mary Tonsager, Paul Toprac, and Neeta Vallab.

I am indebted to you all.

Neil Iscoe
May 1990

Table of Contents

List of Definitions

Chapter 2 – Attributes

Chapter 3 – Classes

Chapter 4 – Hierarchical Decomposition

Chapter 5 – Class Composition

Chapter 6 – Class Association

List of Figures

Chapter 1 - Introduction

Chapter 2 - Attributes

Chapter 3 - Classes

Chapter 4 – Hierarchical Decomposition

Chapter 5 – Class Composition

Chapter 6 – Class Association

Chapter 7 – Related Work

Chapter 8 – Conclusion

Chapter 1 – Introduction

Programmers must have an understanding of both programming knowledge and application domain knowledge to write application programs. But while programming is well enough understood to model and teach, application domain knowledge is not yet well understood, and is codified only in an informal ad hoc manner. Because representations that precisely characterize application domain knowledge do not currently exist, errors are frequently made when gathering and mapping specifications from the informal to the formal.

This dissertation defines a meta-model for application domain knowledge and describes a methodology for its instantiation into domain-specific models. Domain models are representations of application domains that can be used for a variety of operational goals in support of specific software engineering tasks or processes. The meta-model and methodology in this dissertation facilitate understanding and analyzing application areas and eliciting and formalizing software requirements and specifications. The emphasis is on general characterization techniques that can be used to instantiate models from different application domains. Figure 1.1 shows how informal application domain knowledge is instantiated into formal domain models.

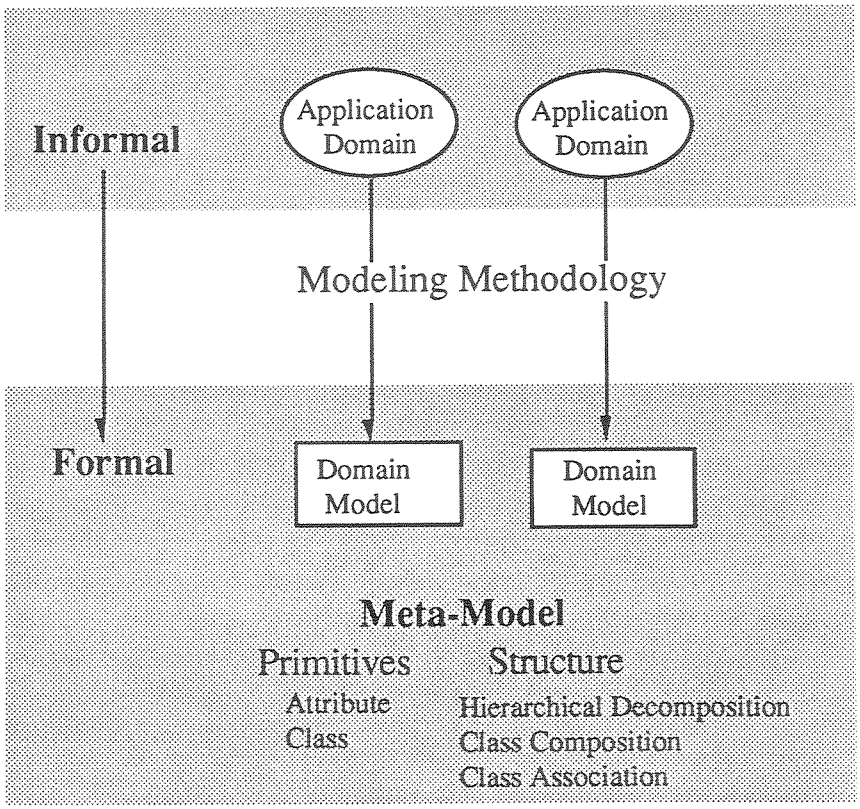


Figure 1.1 — Instantiating Domain Models

Domain models are instantiated by representing specific application domains in terms of attributes and classes. Classes are organized and structured using hierarchical decomposition to create subclasses, and composition and association to construct larger

classes from smaller ones. Examples are given in terms of library, accounts receivable, and other related problems.

1.1. Organization of the Dissertation

This chapter gives the background and motivation for the dissertation research. It outlines the mapping process from specifications to implementation, and shows where problems occur in the process of converting informal views to formal computer programs. A solution paradigm is proposed in which the results from this dissertation and others like it will eventually allow designers, who are neither computer programmers nor domain experts, to construct application programs by declaratively describing and refining specifications for the programs that they wish to construct. This chapter concludes with an overview of the meta-model and modeling methodology.

Chapters 2 through 6 describe the model in more detail. Standard computer data types are semantically bare methods of representation that are unable to capture fundamental concepts in an application domain. Chapter 2 defines attributes as meta-model primitives that capture the semantics of domain properties in terms of scales (from mathematical measurement theory), units, quantities, granularity (from the physical sciences), population parameters (from statistics), and value set transitions.

Chapter 3 defines*classes* that encapsulate sets of attributes, provide for the definition of derived attributes, allow additional operations to be defined, are responsible for object instantiation and deletion, provide for the definition of axioms, and maintain summary statistics on object sets.

Chapter 4 defines hierarchical decomposition; the process of developing a class hierarchy by using *attribute restriction* to *specialize* a class into *subclasses*. Sub (and super) class definitions, along with our interpretations of the terms *inheritance, generalization, specialization, aggregation, and instantiation* are explained in chapter 4.

Chapter 5 defines class composition; the process of creating a new class from two or more other classes by taking the union of their sets of attributes and using domain knowledge to resolve conflicts, eliminate unnecessary attributes, and add new attributes as needed.

Chapter 6 defines class association; the process of creating a new class that establishes a relationship between at least two other classes by taking the union of their naming attributes and using domain knowledge to add new attributes as needed.

Chapter 7 reviews related work, and Chapter 8 summarizes the results of the dissertation.

1.2. The Mapping Process

Creating an application program is a process that begins with a person's conception of an imagined program, and ends with a binary representation of information that can be executed by a machine. This mapping process can be viewed as beginning with a set of requirements or specifications, and ending with a composition of primitive operations that define an implementation. Producing a successful mapping is a process that is intrinsically difficult, usually iterates, and rarely terminates. Furthermore, the stages of the process tend to blur as decisions made in the various stages of implementation inevitably effect and change the original specifications [Swartout 82].

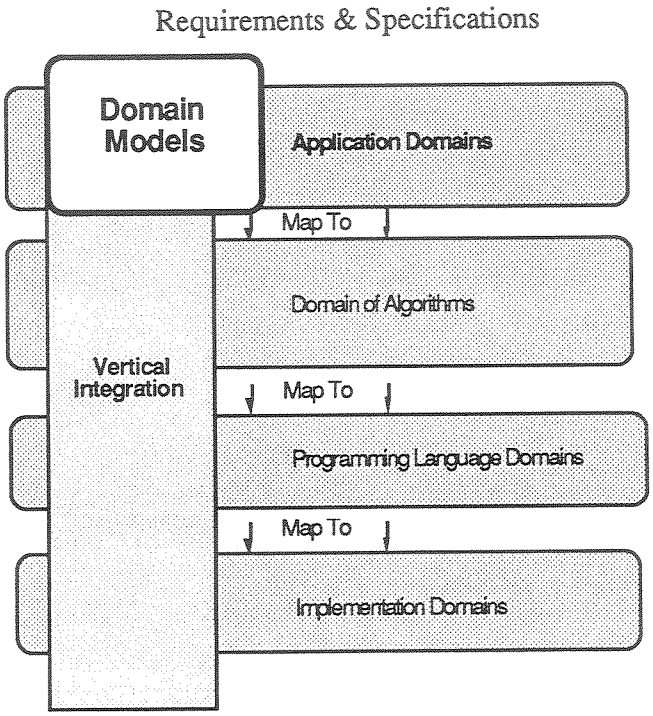


Figure 1.2 — Mapping Requirements To Implementation

Figure 1.2 shows a view of the mapping process as it progresses from requirements and specifications expressed in terms of an application domain through computer science and programming language domains to a final level of implementation. While the figure is a rough abstraction, it provides an outline to which we will refer in future sections of the dissertation.

Computer Scientists are accustomed to classifying information in terms of bare data types such as boolean, integer, real, enumerated, and others that have been mapped up from computer architectures. These types are useful abstractions that provide more power and generality than simple bits. Unfortunately, these types are semantically bare from the point of view of application domain modeling; reflecting the fact that much semantic information is lost in the mapping process.

When a domain is well defined and bounded, it is possible to cleanly separate specifications from implementation. In a structured model of program development[1] code is written to satisfy specifications which are derived from requirements about a particular problem. But since most application domains are neither well defined nor clearly bounded, mapping from the ambiguous world of application requirement and specifications into computer language data types often results in an information loss [Curtis 88].

Information about each of the domain levels is lost between mappings because the information needs of each domain result in different representations. Furthermore, the lack of a formal model to represent information at the application domain level means that semantic information can only be represented in an informal manner. This can create critical problems because high level specification changes can no longer be mapped into the

---

[1] such as that described by [Dijkstra 72], [Hoare 72], [Hoare 73], [Hoare 85],

code that implements those specifications. Consequently, our focus is on modelling application domains, which are shown at the top of Figure 1.2.

The vertical box on the left side of Figure 1.2 is to indicate that vertical integration of the various domains is one way of achieving efficient mapping. Efficiency issues addressed in using vertical integration are discussed in [Hufnagel 89] but are not the subject of this dissertation. Instead, we are concerned with the box labeled "Domain Models," and only attempt to model simple (programming-in-the-small) application domains.

1.2.1. Life Cycle Paradigms

For the last two decades, the waterfall model has been a classic description of the mapping process within software development. The model, which takes its name from the stair step diagram of development stages which resemble a waterfall, was first published by Royce [Royce 70], and further popularized with an expanded version by Boehm [Boehm 76]. Figure 1.2 can also be viewed as a waterfall (e.g., classic sequential) model of software development.

There have been many critiques of the model [McCracken 81], and there is general agreement that new paradigms are needed for software development. Although the basic concept of mapping from requirements to implementation still exists, there are now other life cycle models which more faithfully reproduce the reality of software development. Prototyping [Boehm 84] is the key idea behind most of the approaches. Refinements on prototyping include: operational specifications and transformational systems [Agresti 86a], [Agresti 86b], as well as rapid throwaway prototypes, incremental development, evolutionary prototypes, reusable software, and automated software synthesis [Davis 88]. All of these methods are based around the old software engineering principle that says you always have to build a system and throw it away before you know what it is that you really wanted to do [Brooks 75][2] These models provide mechanisms for more immediate feedback for application developers. They are all attempts to provide the user with an earlier view of a proposed finished product than the more static waterfall model.

The only real change to Brook's statement of fifteen years ago is that system designers will eventually no longer need to be computer professionals. With the popularized use of personal computers, spreadsheets, and simplified data base programs, many people who had previously never touched a computer are now responsible for their own program maintenance. Unfortunately, this creates a whole new set of problems.

1.2.2. Users Must Create Their Own Programs

Users must begin to directly create and maintain their own programs because it is unreasonable to expect that there will ever be enough professional programmers to meet the continually increasing demands for software. The problem is somewhat analogous to the difficulties that occurred with the proliferation of telephones during the early part of this century; it was predicted that if the telephone growth rate continued, eventually every man, woman, and child in the United States would have to become a telephone operator. And of course everyone eventually did. Networking technology eliminated the need for patch

---

[2] *"The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. The only question is whether to plan in advance to build a throw away, or to promise to deliver the throw away to customers.... Hence, plan to throw one away; you will, anyhow."*

boards by solving the procedural aspects of call routing, but placed the burden of declarative specification (i.e. specifying a phone number to be called) on the enduser. A similar solution to the problem of creating and maintaining complete programs within well understood application domains is both necessary and achievable.

1.2.3. Automating the Mapping

Because one of our goals is to remove the application programmer from specific application programming tasks, this general area of research is sometimes referred to as automatic programming. However our goal is to formalize an approach to application domain modeling. As [Barstow 84] has observed, "*... a primary goal of automatic programming research should be the development of a model of domain-specific programming and techniques for building domain-specific automatic programming systems.*"

How has the view of automatic programming changed throughout the decades? It is interesting to realize that in 1958, the term *automatic programming* was used to differentiate the language FORTRAN from lower level assembly and machine languages [Rich 88a]. Through the development of programming languages, computer science has given programmers progressively better tools with which they can map their understanding of program specifications more directly to the computer by using high level constructs instead of machine specific operation codes.

The first step in automating programming was the development of assemblers that could assist in the task of composing these operation codes. Compilers immediately followed assemblers, and were viewed by many as an interesting, though originally inefficient, scheme for allowing programmers to work at a higher level of abstraction than is possible with assembly language. The construction of early compilers was ad hoc, but researchers have changed the construction process from a craft to a science, and have directed research towards specific subjects such as grammar types, parser construction, code optimization and so on. By separating the concerns of compiler construction from the issues of language design, the computer science community has been able to proceed with investigations into substantive language issues without usually having to carry the baggage of compiler or interpreter design. But even as languages reach to higher and higher levels, as [Smoliar 83] points out, "no matter how high the level, it's still programming."

1.2.4. Application Programming

Creating application programs within well understood domains is a *programming in-the-small* process that can be accomplished by people who understand both programming and a specific application domain. Although there is a general belief [Neighbors 84a], [Barstow 84], [Adelson 85], that knowledge of a domain is a prerequisite to being able to write good programs within that domain, the specificity of this knowledge means that in addition to learning programming, programmers must often spend years learning the characteristics of a particular domain [Curtis 88]. This knowledge is used to build application programs, but is rarely codified in a way that allows it to be *well understood* enough to be reused by others.

Even when a domain is well understood, application programming is a difficult process that extends into the real world of human computer users. Most real world domains are

neither well defined nor clearly bounded and consequently large amounts of effort are required to elicit and clarify the users image of an application program. While today's professional programmer is more likely than his or her predecessor to be able to construct and verify the correctness of a particular section of code, the gathering and refining of the specifications for that piece of code is still a black art—a skill which we are attempting to partially automate.

1.2.5. Application Domain Knowledge

Even though definitions vary with the operational context, researchers agree that domain knowledge:

- is necessary to write good programs within a domain [Adelson 85];
- is a requirement for program generators [Barstow 84]; and
- is poorly defined and frequently takes years to gather [Curtis 88].

Software designers implicitly use and reuse application domain knowledge to prevent specification errors, and it is the absence of this knowledge that allows many types of errors to occur. Although domain knowledge is an integral part of many fourth generation and other special purpose languages, it is almost never formally represented and rarely codified in a way that allows it to be well understood enough to be used by others. What then is application domain knowledge? How can it be represented? What kind of errors can it prevent? How can it help in the design and construction of application programs? This dissertation outlines an approach to answering those questions in terms of a new paradigm for application programming.

1.3. Requirements/Specifications

The failure to capture domain knowledge results in a variety of different problems in the creation of specifications. Examples of some of these problems are given in the remainder of this section and are summarized as follows:

- An inadequate understanding of the mapping between specifications and code.
- The failure to recognize errors in specifications.
- Misunderstanding of terms that are operationally defined within a particular domain.
- The difficulty of communicating between domain expert and application designers.
- An insufficient understanding of the implementation level resource tradeoffs that are inherent in a specification.

1.3.1. Specification Errors Result From A Lack of Domain Knowledge

Problems result when a programmer neglects to acquire enough domain experience. One modern software engineering tale concerns the navigational system of the F-16 fighter plane. In this story, a simulated F-16 flipped 180 degrees during a computer simulation of a routine flight from the Northern to the Southern Hemisphere. The behavior happened at the equator, where the plane's navigational program apparently was unable to handle the

situation in which previously decreasing latitudes were now increasing without a change of direction.

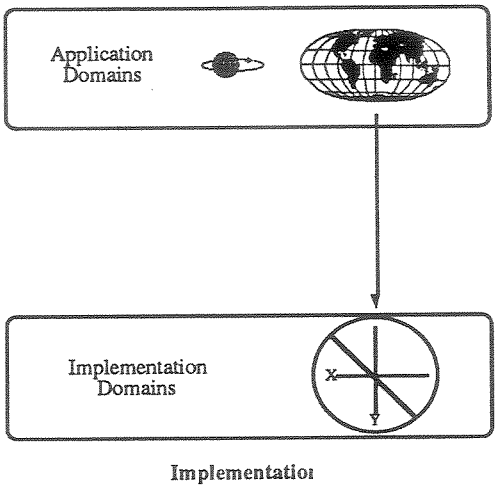


Figure 1.3 — Latitude Mapping Error

What is it about the specification that caused, or failed to prevent, this mistake? While our answer is only conjecture, we assume that if the plane hadn't crossed the equator that there wouldn't have been a problem. Therefore, the error was caused by the failure of the programmer to recognize that there are no negative latitudes. This might have been caused by the programmer visualizing the world as mapped to a Cartesian Coordinate Plane where the y-axis was latitude, and a latitude of zero represented a position at the bottom of the map. Other interpretations are possible, but the overall point is that errors can result when domain knowledge is lost in the mapping process.

1.3.2. Reasonable Requirements Are Often Wrong

The next example illustrates a subtle problem of the dangers of believing specifications as they are given without using a broader domain-knowledge view to interpret those specifications. The following two requirements are taken from the statement of the definition of a library system [Kemmerer 85] (further discussed in Appendix A):

> ***Requirement 1***
>
> *All copies in the library must be available for checkout or be checked out.*
>
> ***Requirement 2***
>
> *No copy of the book may be both available and checked out at the same time.*

These requirements seem reasonable, and when axiomatized in a specification system such as Larch allow theorems such as the following to be proved:

If a book is not checked out and is not available, then it is not a library book at all [Wing 87].

$(\sim checkedOut(l,b) \wedge \sim available(l,b)) \Rightarrow b \notin allBooks(l)$

But books that have been lost, or stolen, or damaged, can violate requirement 1. In our view, the specification is wrong because it was based on a library model where critical knowledge about the domain was not captured. Consequently, proofs based on this requirement, while correct within the bounds of the proof system, are results that would not be acceptable for librarians.

Our concern is with the levels of the application domain that map to the real world. We believe that a formal model of a library application domain would prevent errors such as these and others pointed out in [Wing 88]; thus, we are concerned with creating a model of an application domain that will avoid, among other problems, specification errors of the type demonstrated above.

1.3.3. Terms Are Ambiguous Without A Domain Context

Within any particular application domain, a variety of domain-specific terms are always used. Some domain analysis techniques [Neighbors 84a], used these domain-specific terms as the basis for a grammar which is then considered to be a model of a domain. While this is a useful and successful technique, it is limited to any particular domain of application. Our approach is to attempt to build general models that allow otherwise ambiguous terms such as "2/10 net 30" to be incorporated directly into constructs of the model.

1.3.4. Semantic Information is Required to Solve Conflicts

The following example of a specification problem concerns the ambiguity of simple terms without clarification by domain-specific use.

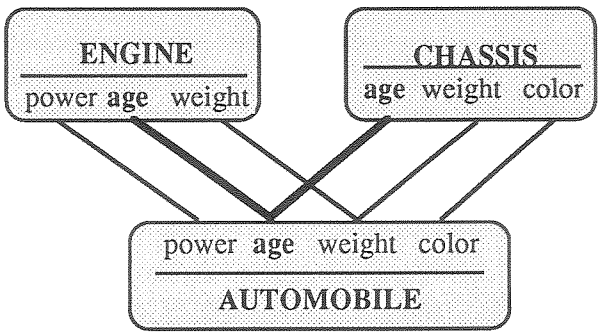


Figure 1.4 — Determining the Age of an Automobile

In Figure 1.4, an engine has a power rating, an age, and a weight. A chassis also has an age and a weight, but does not have power and does have a color. Clearly, an automobile should have at least the properties *Power* (inherited from Engine), and *Color* (inherited from Chassis). But what about *Weight* and *Age*? Assuming that the ages of *Engine*, *Body*, and *Automobile* are all measured in the same unit (say years), *automobile age* could be calculated in at least the ways shown in Figure 1.5.

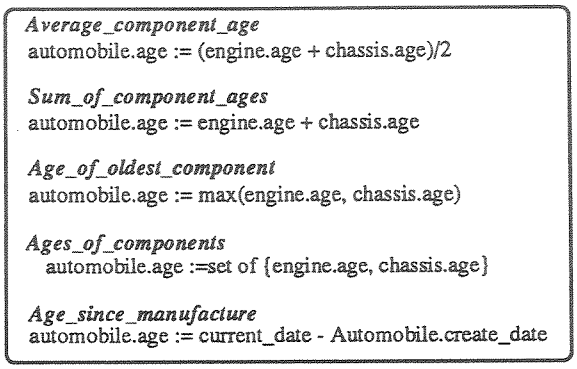


Figure 1.5 — Some Interpretations of Age of a Car

Within the domain of car buyers, most people would say that automobile age would best be calculated by the *Age_since_manufacture* operation that uses the date that items are assembled to form the basis for the age. But what about other domains? For example, the following are all reasonable criteria for engine age in the domain of engine repair:

- Age of automobile engines is measured in miles .
- Age of automobile engines is measured in years.
- Age of boat and airplane engines is measured in hours.

Domain knowledge is used to answer this potentially ambiguous question about the age of an engine. This is one of the types of information, normally lost in the creation of an application program, that a domain expert can capture in a domain model.

1.3.5. Requirements and Specifications Change

Finally, requirements change. Even when requirements and specifications are correctly gathered, refined, and then transformed correctly into executable code, the nature of most real world applications is that requirements and specifications mutate throughout the existence of a program. These modifications are neither caused by the sometimes mercurial nature of endusers nor by sloppiness in the original design process, but are the result of natural occurrences such as workload modifications, environmental changes, new technologies, economic disturbances, legal mandates, and so on, that remove old requirements and create new ones.

Automating the transformational mapping of specifications to code and allowing the full program specification to be stored in terms of the domain will mean that users can change specifications and then regenerate their application program.

1.4. Solution Paradigm

From the perspective of program development, the eventual goal is to allow designers, who are neither computer programmers nor domain experts, to construct application programs by declaratively describing and refining specifications for the programs that they wish to construct.

Figure 1.6a is an overview of the solution paradigm as described in this dissertation and Figure 1.6b is a more detailed schematic. The top right portion of Figure 1.6b shows a representation for domain knowledge that can be used as the basis for generating domain-

specific application program generators. This dissertation focuses on modeling domain knowledge (the top rectangle of the figures). In particular, the focus is on a precise and operationally useful set of modeling techniques that will facilitate the elicitation of domain knowledge from domain experts.

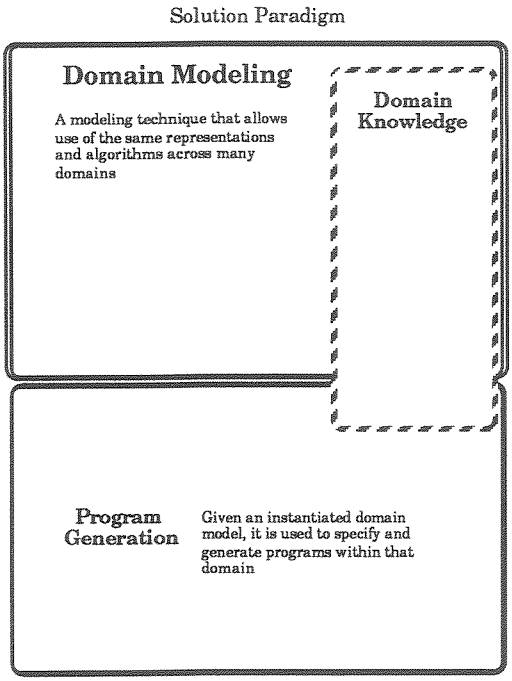


Figure 1.6a — Solution Paradigm

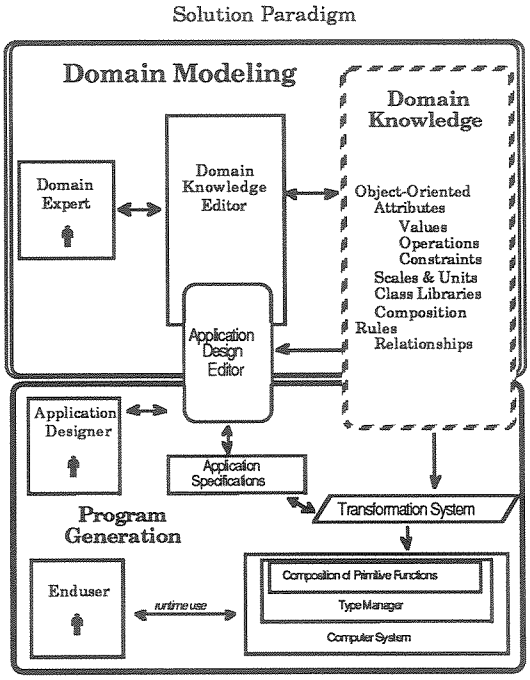


Figure 1.6b — Agents & Modules

1.4.1. System Users or Agents

There are three types of human users or agents that are of interest in this paradigm. Figure 1.6b shows these users as domain experts, application designers and endusers. This dissertation is primarily concerned with domain experts. Domain experts are people who, through years of experience, understand an application domain [Curtis 88].

Although domain experts may have had experience with computer systems they are presumed to be neither computer scientists nor programmers. Instead, they understand application domains through training and experience. Domain experts are library scientists, inventory control managers, retail specialists, and so on. The purpose of the domain model is to capture their knowledge of the application domain. The modeling of that knowledge is the subject of this dissertation.

Endusers are the final users of an application program as shown on the lower left of Figure 1.6b. Although endusers use application programs, this dissertation describes a system designed to be used primarily by domain experts and secondarily by application designers. Consequently, within the context of this dissertation system a user is actually a domain expert or application designer. In a management information systems context, end users are the clerks, secretaries, and perhaps managers, who directly use an information system. Within the context of real time domains such as telephony, endusers are the people who access or use the system.

Application designers will vary in technical sophistication, and range in expertise on a continuum from domain expert to enduser. We use the term to distinguish them from experts who are presumed to have an all encompassing view of the domain, and from endusers who are the class of users at the end of the development chain. An application designer might be a librarian in Peoria, Illinois who knows enough about a library to specify their own system. Or they might be a receivables supervisor who is intimately familiar with the operations of their company. From an operational view, application designers are anyone who has a thorough enough understanding of their needs to be able to interactively specify a system.

1.4.2. Dissertation Focus is on Domain Experts

This dissertation is primarily concerned with the aspects of domain model and the modeling methodology that facilitate the elicitation and structuring of knowledge possessed by domain experts. Throughout the dissertation, examples are given in terms of their expected use by domain experts. However, since an ultimate result of the paradigm is to provide for automatic code generation, we are concerned with application designers and endusers. Consequently, we describe aspects of the model that effect them in places. Unless otherwise specified, the users of the system are considered to be domain experts.

1.4.3. Program Generation

The lower portion of Figure 1.6b shows a domain-specific transformational program generation system. Application designers interact with a design editor to create a set of application specifications. Given a set of application specifications, a transformation system creates a composition of primitive functions to be executed by a type manager that operates within a computer system. This portion of the system completes the mapping from the application domain to the executable code. Research of this type is sometimes called *narrow domain automatic program generation* [Rich 88a], and is further described in Chapter 5 on related work. Our meta-model and methodology are intended to serve as a basis for an interactive support system for development of domain-specific application generators. The emphasis is on the top portion of Figure 1.6b. Although the creation of executable programs by performing a series of transformations on application specifications is not the subject of this dissertation, we have done work in this area [Iscoe 88a] with Batory's [Batory 88] database domain model.

1.4.4. Domain Models and Meta-Models

Throughout this dissertation we use the terms meta-model, domain model, and modeling methodology. Figure 1.1 illustrated our view of these terms. The meta-model consists of a set of primitives that can be instantiated to produce a domain model within a particular domain. The modeling methodology is the methodology that a domain expert uses to instantiate a particular domain model using attributes, classes, inheritance, and composition. The meta-model and its associated methodology for instantiating domain models is sometimes referred to as the modeling technique.

1.4.5 Domain Model Instantiation

Instantiating a domain model is a process requiring an expert with a comprehensive understanding of the domain. Because the domain expert has mentally traversed the depth and breadth of a design space, they have a well defined conceptual model of an application domain [Curtis 88] which allows them to instantiate domain models in fewer design iterations than would be required by someone with less domain knowledge. This expert knowledge is acquired through experience and is the way that much design is performed. As Alexander states in reference to architectural domains:

> *A moment's thought will convince us that we are never capable of stating a design problem except in terms of the errors we have observed in past solutions to past problems. Even if we try to design something for an entirely new purpose that has never been conceived before, the best we can do in stating the problem is to anticipate how it might possibly go wrong by scanning mentally all the ways in which other things have gone wrong in the past. [Alexander 64]*

While structuring a model is inherently an iterative process, by assuming domain expert sophistication we can present a methodology with more of a top-down than a bottom-up flavor. Consequently, certain aspects of our approach differ in both definition and spirit from other object-oriented approaches.

1.5. The Dissertation Research

The creation of a meta-model and associated modeling methodology was in itself an exploratory and iterative procedure. In our research, the process began with an examination of existing models and modeling approaches (summarized in Chapter 7). The meta-model was constructed in the manner that Davis [Davis 88]describes as system evolution. That is, a basic model of attributes and classes was developed, and then an iterative process of prototype development and model refinement was pursued. Several computational platforms were used to prototype portions of the model and methodology.

After initial attribute and class definitions were completed, an iterative series of steps were taken in which Macintosh screen and menu interfaces were designed from the viewpoint of domain expert use. These interfaces were used with examples to further refine the model definition which was then reprogrammed on the Macintosh. Some of the Macintosh screen dumps used to develop the meta-model are shown as figures in Chapters 2 and 3.

Using a different computer system, Prolog was used in two different ways. As more fully described in [Iscoe 88a], a system was constructed to assess the utility of transformational implementation of executable specifications based on high level requirements. Additionally, unit conversion and other aspects of unit domain information described in Chapter 2 were prototyped in Prolog.

In order to resolve ambiguities that might be present from a text explanation of the algebraic meta-model definition, portions of an attribute grammar [Knuth 68] were defined to more fully describe semantics such as attribute addition. Attribute grammars have been used by Wills [Wills 87]to define a system for reverse engineering) and by a variety of other researchers. The attribute semantics described in Chapter 3 and summarized in

Appendix C represent the beginnings of a compositional calculus. Appendix D contains summaries of the meta-model definitions.

1.5.1. Example Domains

The meta-model and methodology generalize across transaction-oriented business application domains. This generalization is illustrated with examples for a library system and an accounts receivable system. Library systems are computer systems that keep track of and provide for the check-out and return of books. The library example, which is summarized in Appendix A,was selected because of its general use within the software engineering community [Wing 88].

Accounts receivable systems are responsible for the management and control of collecting customer balances. The accounts receivable (AR) problem was selected to assess the generality of the modeling approach. An independent validation of the AR model was performed by a member of The Center for Strategic Technology Research at Andersen Consulting, the consulting branch of a Big Six accounting firm.

Other example problems such as the the automobile composition problem [Cardelli 84] [Cardelli 85]were chosen as a way to illustrate the capturing of application semantics by composition are used in the dissertation to illustrate specific features of the meta-model.

Library systems and accounts receivable systems are commonly treated as completely different types of application programs. They exist for different purposes, they are written for different applications, they are sold as different products, and the domain experts are different people because these are different application domains. Not only are they different application domains, but even within a domain like accounts receivable, scores of both custom and generalized programs exist which perform different functions in a variety of computational platforms.

This dissertation presents a generalized approach – the meta-model – that can be used to model aspects of both of these application domains. In addition, the same overall model can be used, with few additional extensions to model generalized rental systems.

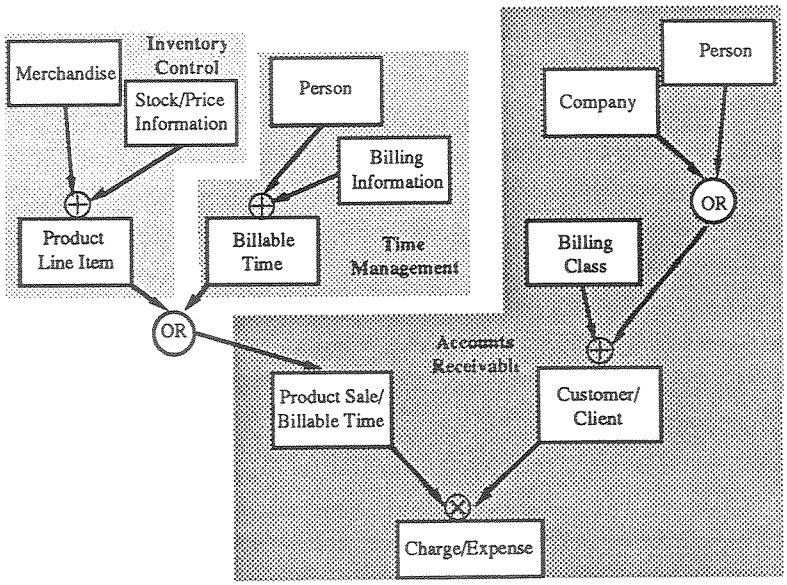


Figure 1.7 — Generalized View of MIS Transaction Application Domains

Figure 1.7 shows a high level view of the entities and associations within business application domains. Within these types of application domains, there are entities that can be added and deleted according to customs and policies of an organization. In addition, there are policies, operating procedures, and laws that effect the association between those entities. Policies and procedures that govern the addition and deletion of entities and the creation and removal of associations between those entities constitute a large portion of the specifications of an application domain. This dissertation will show how this information is modeled within the terms of the modeling methodology.

In Figure 1.7, the box labeled invoice represents a transaction in which a business or person purchases a product or service. This figure can be viewed in a variety of different ways. Focusing on the left side of the picture and assuming that a product is selected, the figure illustrates an inventory control system. However, if instead of selecting a product, the domain expert selects a service such as billing hours, then the left side of the diagram illustrates a professional time tracking system. This dissertation uses as an example the right hand side of the picture, which represents a generalized accounts receivable (AR) program. In general, AR programs are those that keep track of payments for goods or services. A variety of conditions may be attached to those payments. For example, goods sold on a *"2/10 net 30 basis"* are materials that need to be paid for in 30 days, and for which a 2% discount is given if payment is received within 10 days. Additional interest might also be charged for payments made after 30 days.

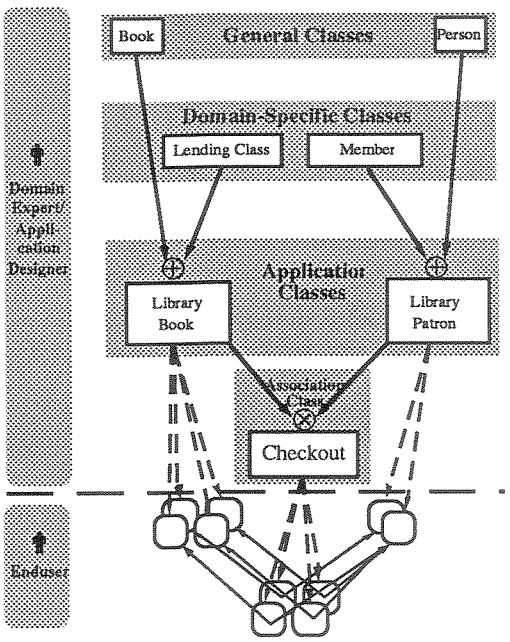


Figure 1.8a — Library System

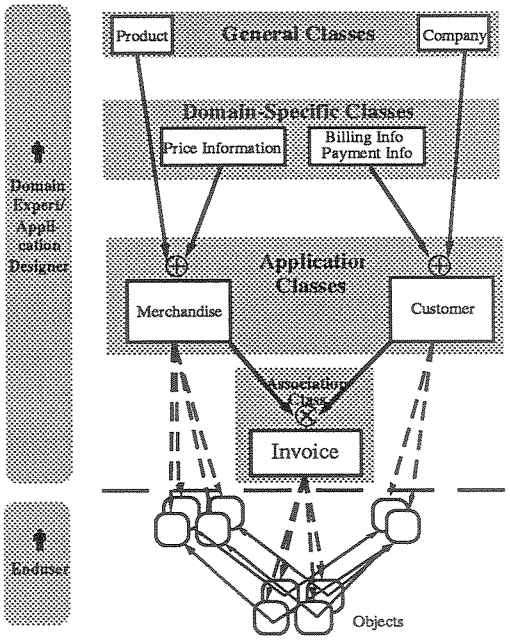


Figure 1.8b — AR System

1.5.2. Programming-in-the-small

The term *accounts receivable* is used by different people in different ways. It could mean a billing system, a point of sale system, or an inventory system. In large operational environments, these programs rarely exist in isolation. Within a software engineering context, examination of a small, perhaps out of context, but well-understood domain is

called "programming-in-the-small." In contrast, programming-in-the-large problems are generally poorly defined, cover a number of domains, and have a host of other problems which exclude them from consideration in our research.[3]

Even within limited domains, it is possible to acknowledge and interact with other systems. Figure 1.9 shows different ways that an invoice makes cash balances in various parts of even simple accounting systems. Since this dissertation addresses only programming-in-the-small, any application of the meta-model and the associated methodology to programming-in-the-large problems is an open research problem.

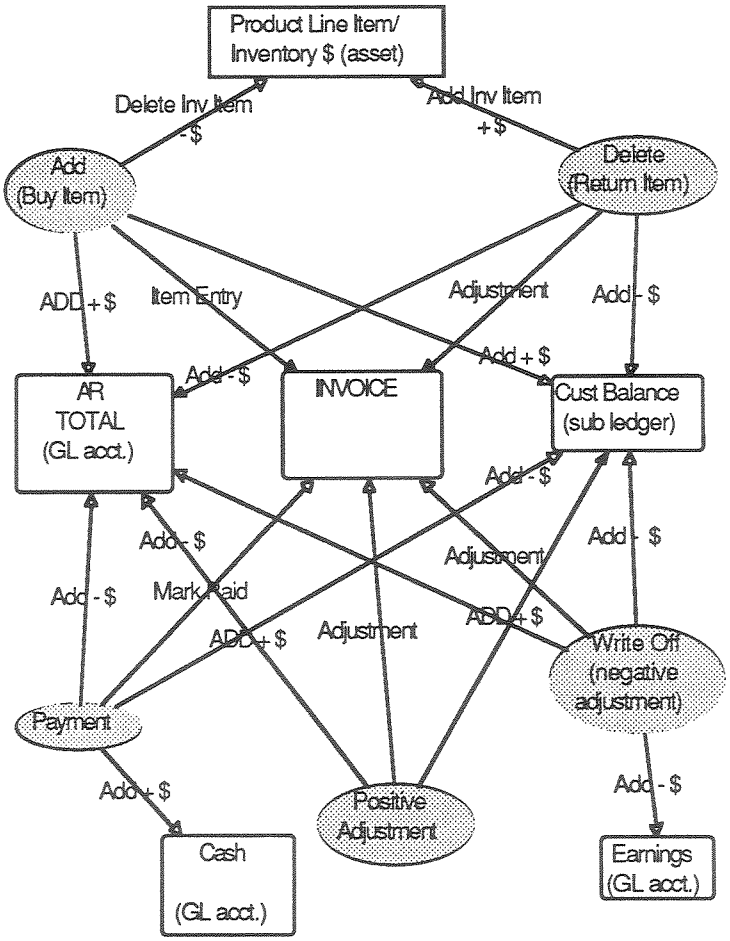


Figure 1.9 — Simplified View, Business transaction cash flow

1.5.4. Previous Research

In previous research [Iscoe 86], a lower level, detailed system was constructed to elicit and implement the lower level specifications associated with attributes such as field length, types of characters to be allowed in a name, and so on. The goals of that research were much more narrow but consequently more completely implemented. That system was capable of generating FORTRAN, Pascal, Cobol, and Basic code which made calls to the

---

[3] A complete discussion of these problems is not relevant to this dissertation, but good discussion of programming-in-the-large problems can be found in [Brooks 75], [Wegner 84], [Brooks 87], [Curtis 87].

type manager that handled all of the detailed enduser screen interaction for text oriented screens.

1.5.5. Formality and Notation

While some researchers believe that informal information [Biggerstaff 89] is a necessary part of their domain models, we have formalized most portions of the meta-model presented in this dissertation. As Davis states in his analysis of life cycle methods:

> *Where do you apply a formal technique and where do you apply something else? The answer is: use a formal technique when you cannot afford to have the requirement misunderstood. [Davis 88].*

While it is true that certain aspects of requirements and specification information are not yet well enough understood to be formalized into a domain model, we see lack of formalization as an obstacle to be overcome. Although excessive formality can sometimes stand in the way of completion of "scruffy" operational goals, we have tried to make our approach as formal as possible by providing definitions and, where necessary, additional semantics in the form of an attribute grammar.

Formality, however, should not be confused with rigid rules of syntax. It was not our intent to create *yet another* object-oriented language. Instead, portions of this model can be used in already existing and future languages.

1.5.6. The Model

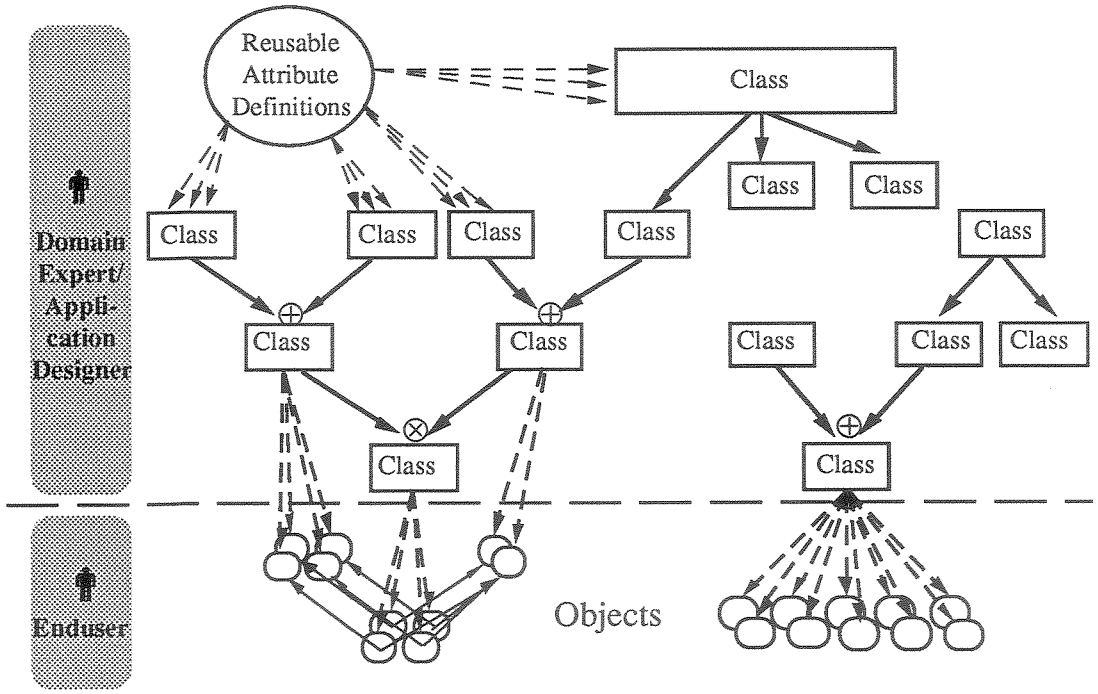


Figure 1.10 — Overview of the Model

Figure 1.10 shows an overview of the model showing attributes, classes, composition (indicated by the symbol ⊕), and association (indicated by the symbol ⊗). While many object-oriented and knowledge representation schemes use these concepts, our

methodology is more definitional, strongly typed, and structured than most other approaches. This is possible because our design decisions were predicated on the notion that the model will only be instantiated on well understood domains. We have traded structure for exploration. The emphasis in this methodology is on providing a meta-model that domain experts can use to record their knowledge instead of an exploratory system in which novices make discoveries about an application domain. This tradeoff allows us to avoid many ambiguities and much confusion such as that introduced by overloading the meaning of the term *IS-A link* [Brachman 83].

As was explained in section 1.1, Chapters 2 through 6 define the elements of the meta-model and give examples of their use. The meta-model begins with attributes which are defined in chapter 2, and continues with classes defined in chapter 3. Chapter 4 explains and defines structuring through hierarchical decomposition, while chapter 5 and 6 define the class construction operations composition and association.

Chapter 2 – Attributes

2.1. Introduction to Attributes

Attributes are type descriptions that characterize value sets within application domains. In most programming languages, values are classified in terms of boolean, enumerated, integer, real, and other data types that have been mapped up from computer architectures. Unfortunately, standard computer types are semantically bare methods of representation that are unable to capture fundamental concepts in an application domain. Although many specification and database languages go beyond basic programming language data types by characterizing attributes in terms of set and logical formalisms, these approaches still neither capture the underlying semantics of an application domain nor provide structuring mechanisms for eliciting this information.

This chapter introduces an approach for representing attributes in such a way that semantic information is captured with a generalized formalism that can be instantiated in a domain-specific manner. Attributes represent information that ordinarily is not maintained throughout the development and evolutionary lifecycle of a program. Instead of a representation mapped up from computer architectures, an attribute is a representation that maps downward from application domain concepts. It preserves the constraints imposed by an application domain and facilitates the transformational implementation of program specifications. Figure 2.1 is an attribute oriented view of the transformational mapping process from application domain to implementation.

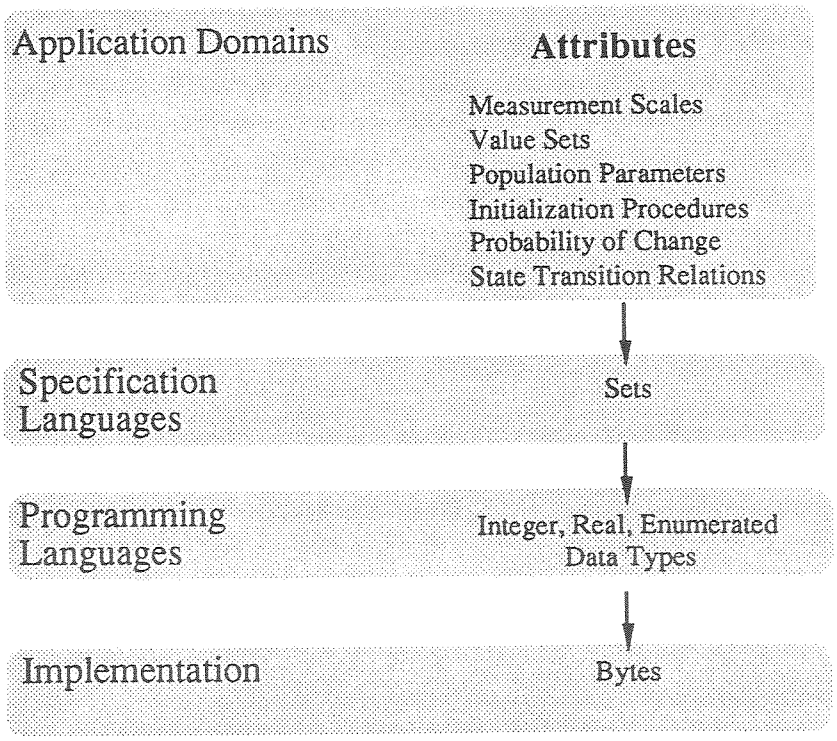


Figure 2.1 — Mapping from Attributes to Bytes

2.2. Formal Definition

Attributes encapsulate state and state change information about the measurement of entity properties. This information is expressed in terms of scales, units, and granularity that constrain the sets of allowable attribute values and the transitions on those values. Changes in attribute state are described by state transitions within value sets which are pairs of values (old_state, new_state).

The sets of possible attribute values and the relations on those values are further constrained by population parameters that assist the domain expert in structuring domain information and can be used by a runtime program to infer default information, display information, and maintain integrity constraints. An initialization procedure maintains information about default values and uses value set distributions to determine the defaults that should be displayed to the enduser. A probability of change is associated with each attribute to help the domain expert with classification decisions (discussed further in Chapter 4) and to aid in future transformational implementations.

Definition 2.1 is the formal definition of an attribute within the meta-model and is the subject of the remainder of this chapter.

An attribute, A, consists of :	
a unique name $\mathcal{N}$(A)	
a measurement scale $\mathcal{M}$(A) and (when appropriate) unit & granularity	[Section 3.4-5]
a set of values $\mathcal{V}$(A)	[Section 3.3]
a set of population parameters $\mathcal{PP}$(A)	[Section 3.6]
an Initialization procedure $\mathcal{I}$(A)	[Section 3.7]
a probability of change $\mathcal{PC}$(A)	[Section 3.9]
a state transition relation $\mathcal{R}$(A), $\mathcal{V}$(A) x $\mathcal{V}$(A) $\supseteq$ $\mathcal{R}$(A)	[Section 3.10]

Definition 2.1— Attribute

2.3. Scaling Theory and Value Sets $\mathcal{V}$(A)

The typical programming language characterization of attributes as basic data types such as integer and real numbers, or the specification language characterization of attributes as sets does not capture the semantic information required to eliminate specification problems such as those discussed in Chapter 1. Furthermore, simply achieving the mapping shown in Figure 2.1 is not sufficient, because it is critical that a systematic means of specifying value set restrictions—a modeling methodology—be made available to the domain expert.

Attributes represent the natural value restrictions of an application domain instead of relying on simple data types. In creating a definition of an attribute, we pursued a more formalized approach instead of relying on text annotations or other ad hoc informal methods. Scaling theory is a mathematical methodology that provides the theoretical foundations for the measurement and classification system used for value sets in the meta-model and methodology.

Mathematical scaling theory is used for classification in a variety of fields and is not a new field of study. Most of the results in use today have been unchanged for over 30 years. Furthermore, their characteristics are so well understood so that we will not prove

their properties in this dissertation. The following references are some of the more well known contributors to scaling theory: [Stevens 46], [Coombs 53], [Coombs 60], [Suppes 58], and [Torgerson 58]. Generalized summaries of scaling theory include: [Blalock 72], [Stamper 73], [Stevens 74], and [Curtis 80].

Scaling theory was developed as a methodology for describing measurements of psychological, social , and physical information. Widely used in political science, economics, psychology, and other fields, scales are designed to characterize properties. Consequently they serve as ideal vehicles for capturing, structuring, and storing attribute semantics within a domain model. Scales are used in a domain model as a means for characterizing attribute value sets by restricting the values, the relevant population parameters, the initialization procedures, the transition relations, and the transformational implementation of those value sets in a manner that is both formal and constrained enough to capture the necessary domain semantics to achieve our operational goals.

The next few sections describe the four basic scale types[1] as they relate to this modeling technique. The discussion begins with the most basic scale type, the nominal scale. The qualitative ordinal scale is then introduced and followed by the more quantitative interval and ratio scales. Each of the scale types is defined along with the restrictions that they place upon a value set, their mathematical properties, the population parameters that are used for their characterization, and the allowable transitions upon their value sets.

2.3.1. Nominal scales

A nominal scale is the most basic measurement scale, and can be defined as follows:

A nominal measurement scale m is a set of categories $C(m) = \{C_1, \ldots C_n\}$.

Definition 2.2 — Nominal Scale

In use, a nominal scale can be thought of as a partition of a set of items U into a set of subsets $A_1, A_2 \ldots A_n$. These subsets are the categories of the scale, where a category[2] is simply a grouping of items. The semantics of an application domain are maintained by creating categories in such a way that items to be categorized are as homogeneous as possible within a category and as heterogeneous as possible between categories. The modeling methodology presumes that domain experts have the knowledge to create these categories.

A nominal scale is the most fundamental scale; all scales are nominal scales and possess, at least, the characteristics of nominal scales. Examples of nominal scales abound and map cleanly to the notion of enumerated type. For example, a nominal scale to categorize automobile color selections might be the colors red, yellow, green, and blue:

---

[1] In this dissertation, we introduce only the four most basic scale types. Although these are sufficient to illustrate the modeling methodology, other scale types are discussed in the literature such as logarithmic and partial order scales. See [Labovitz 70] and [Labovitz 72] for an additional discussion of some other issues in scaling theory.

[2] The term category is used in the sense of group or equivalence class and has no relation to mathematical category theory.

```
Colors : Nominal_scale (Red, Yellow, Green, Blue);
    my_truck.color := Blue;
```

The important population parameters of nominally scaled attribute value sets are proportions, percentages, and ratios, and can be calculated for any finite set U in the following manner:

proportion A_i is $|A_i|/|U|$
percentage A_i is $|A_i|/|U| \times 100$
ratio A_i to A_j is $|A_i|/|A_j|$

Equality and inequality are the only natural mathematical relations among items that have been placed in a nominal scale.

2.3.2. *Library Example*

In the library example (Appendix A), two of the constraints were:

All copies in the library must be available for checkout or be checked out.

No copy of the book may be both available and checked out at the same time.

The first constraint states the two categories for a nominal scaled attribute that indicates the checkout_status of a book.

```
checkout_status: Nominal_scale (checked_out, avail_for_checkout)
```

The second constraint is a standard property of a nominal scale, (i.e. a nominal scale is a partition of a set of items U, where each item is contained in one and only one class A_i from the partition $\{A_1,...A_n\}$). Assuming that the preceding constraints are correct, [Wing 87] shows that:

If a book is not checked out and is not available, then it is not a library book at all

$\sim checkedOut(l,b) \wedge \sim available(l,b) \Rightarrow b \notin allBooks(l)$

The problem with this conclusion is that it does not address the cases of lost, missing, or stolen books. This is not the fault of Larch[3], but rather illustrates the danger of a literal interpretation of specifications without the benefit of domain-specific knowledge–semantic information that is captured by a domain model. A more appropriate scale is:

```
Check-out_status:
    nominal scale (not_checked_out, checked_out, lost, missing_stolen)
```

This scale takes into account lost or stolen books. This example continues when transitions between states are discussed later in this chapter.

2.3.3. *Ordinal Scales*

Nominal Scales are used when there is no ordering information about the categories in a scale. *Ordinal Scales* are nominal scales for which a strict ordering between categories is

---

[3] Larch is the specification language used by J. Wing in this example.

obtained by ranking the categories according to the relative value they possess of some characteristic as defined by the scale.[4]

An ordinal scale is a nominal scale in which a total ordering exists among the categories C_i.

Definition 2.3 — Ordinal Scale

Mathematical relations of interest in ordinal scales include the equality and inequality relations of nominal scales as well as the following additional relations:

>>– ranked higher than

<< – ranked lower than

Given that A >> B >> C, one has no information about the numerical differences from A to B, B to C, or A to C. Although rankings on ordinal scales may be obtained in a variety of ways, an ordinal scale provides no information about the magnitude of differences between categories of items. Continuing the automobile color example, colors can be placed on an ordinal scale of perceived warmth as follows:

Colors : Ordinal_scale (Red >> Yellow >> Green >> Blue)

2.3.4. Quantities: Units and Granularity

Nominal and ordinal scales are often called qualitative measures because they do not measure properties in terms of numerical units. This is, of course, why addition and subtraction are not valid operations on these scales. The following definition of quantity is required to introduce the more quantitative interval and ratio scales.

A *quantity* is defined in terms of:

- a unit defined in domain-specific operational terms
- a measurement granularity, that is the highest degree of precision to which a unit is normally expressed within a domain.

Definition 2.4 — Quantity

For example, in a domain such as a law office, the quantity *time* has a unit of an hour and a measurement granularity of quarter hours (granularity 0.25) or tenths of hours (granularity .01).

The selection of unit and granularity is a domain-specific decision that is based on both tradition and technology. The quantity *time*, for example, is measured in unit minutes for long distance telephone call (with the granularity determined by the long distance carrying company), unit seconds for football games (with a granularity of 1), unit seconds for Olympic events (with an increasingly finer granularity determined by the timing devices) and unit millennia for geologic transformations.

Some examples of quantities expressed in terms of their units are: cost in dollars, distance in miles, and weight in pounds. Later in this chapter, derived quantities will be

---

[4] As was mentioned, partial order and other similar scales have been defined but are not used in this model.

introduced; these include more complex quantities such as speed in miles/hour, salary in dollars/hour, salary in dollars/year, and salary in dollars/academic-year.

Choosing a unit can have many consequences. For example, human age is usually discussed in unit years (with a granularity of 1) and is calculated by subtracting birth date from the current date. Although years are a measure normally sufficient for most human purposes, a unit granularity of one year fails to give complete information in cases where a finer granularity is needed (such as at a pediatrician's office). In the case of young persons, age in unit years (with a granularity of 1) doesn't take into account that ages of:

- Children under 18 months are generally measured in unit months.
- Children under 3 months are generally measured in unit weeks.
- Children under 1 week are generally measured in unit days.

Using an age derived from birthdate where the measurement granularity of the base time unit is day, it is simple to calculate ages based on days, weeks, months, or years. Note, however, that when age is calculated from the leap year corrected difference between current date and birth date that it would be incorrect to express the resultant age to a finer granularity than day. This issue will be further discussed in Chapter 3 in the section on derived attributes.

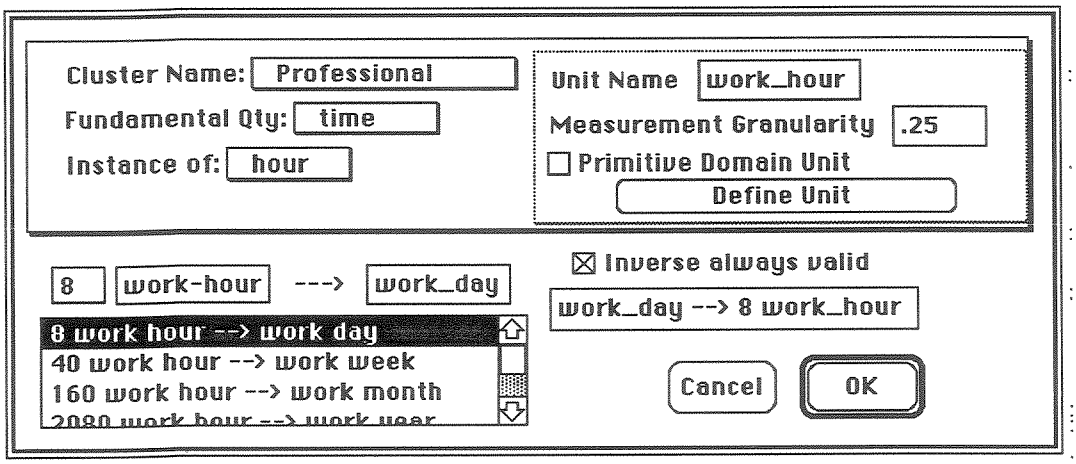


Figure 2.2 — Defining a Work-hour

2.3.5. Measurement Granularity and Precision

The rules of measurement and measurement combination are intimately tied to the physical characteristics, applicable technology, and customs of any particular application domain. *Measurement granularity* is determined in a domain-specific manner by the instruments available for measurement, the enduser perception of the unit, and the expected use of the item being measured.

The effects of measurement granularity can be observed in many systems. For example, in most professional time and billing systems, an hour is a unit of billable time. But in some offices, hours are measured in ten sub units (a granularity of tenths of an

hour), while in other offices hours are measured in four sub units (a granularity of a quarter hour). This seemingly trivial difference has major effects on how a system is used. A five minute phone call might be charged as a billing unit in a tenth of an hour system, while the same call might not be billed at all in a quarter hour system. Similarly, a twenty minute call might be billed as .50 hours or .25 hours in a quarter hour system, or in a tenth of an hour system as.30 hours or .40 hours. Even in systems that are not automated, the choice of granularity can have major effects on operations, revenues and a client's perception of correctness.

The rules of measurement granularity are generally either ignored in most computer system implementations, or captured in an arbitrary or ad hoc style. A goal of the meta-model and domain modeling methodology is to capture units and measurement granularities of quantities and to provide consistent enforcement of measurement rules and operations.

The physical sciences give us many measurement rules that can be maintained by all domain models. For example, freshman chemistry and physics students quickly learn that in order to properly interpret a measurement, it is incorrect to express an answer to more significant figures than are justified by the accuracy of the observed data. In basic lab work one learns that measurements should never be expressed to more significant figures than are justified by the accuracy of the algorithm, the accuracy of the original measurement, or by the purpose to which they are to serve. Furthermore, derived information should never be expressed at a finer level of granularity than is justified by its component parts, the algorithms used to compute it, or by its original level of measurement granularity. The rules of measurement precision are rules that are easy to apply if the initial measurement information is captured. Furthermore, if this information is maintained, these integrity rules can be computed and enforced by an automated system. These rules will be shown to be of particular importance in the next chapter in the sections concerning derived attributes and attribute composition.

2.3.6. Unit Analysis

Although unit checking is a well understood concept, it is frequently ignored at the implementation level of an application program. The necessity of having identical units for both sides of an equation is so fundamental that, in practice, obvious units are often omitted from software specification documents and sanity checks are rarely conducted on specifications. For example, ambient air temperatures in the United States are generally recorded in Fahrenheit while ambient air temperatures in Europe are recorded in Centigrade. Frequently, the only time that quantities are introduced is when there is a perceived danger of ambiguity in the interpretation.

The problem is that when the magnitudes attached to these units are mapped from a domain into an implementation, the semantic information represented by the quantities is lost. Consequently, mistakes such as the F16 example described in chapter 1 often occur in actual practice. These problems could be avoided if higher level semantic information about units, quantities, and granularity were maintained. A domain model enforces these restrictions.

A quantity and magnitude are required to completely describe a value on an interval scale. The next two sections introduce interval and ratio scales and present a formalized way to preserve this information.

2.3.7. *Interval Scales*

Interval Scales are ordinal scales that categorize items according to their position on a scale of standardized intervals or units.

> An interval scale is an ordinal scale that has an associated quantity and assigns a unique multiple (called the magnitude) of the measurement granularity of the unit of the quantity to each category.

Definition 2.5 — Interval Scale

Interval scales have all the properties of ordinal scales, where the ordering is induced by the magnitude, but in addition, the classification of items by magnitude makes it meaningful to apply addition and subtraction functions to items classified according to the scale. The additional population parameters for value sets characterized by interval scales include the mean, median and standard deviation as well as those parameters previously mentioned.

Continuing the previous example, colors could be placed on an interval scale that measures wavelength in nanometers.

Colors : Interval_scale (wavelength in nanometers)
Red 640,
Yellow 580,
Green 520,
Blue 480.

Not all interval scales have the same properties. Consider a United States weather forecasting example where the the unit is Fahrenheit degree, and the unit granularity is one. Although the interval scale Fahrenheit allows the operations addition and subtraction, it does not allow multiplication and division because the Fahrenheit scale does not have a non-arbitrary or absolute zero point. Consequently, it is impossible to make meaningful statements such as "*temperature X is twice as hot as temperature Y*" using a Fahrenheit scale, or to make any other type of ratio comparisons within the scale. This is because the zero in an interval scale has no real meaning within the application domain. Interval scales in which it is possible to meaningfully discuss such comparisons are called ratio scales.

2.3.8. *Ratio Scales*

Ratio scales are interval scales that have an absolute or non-arbitrary zero point. A *non-arbitrary zero*, sometimes called an absolute zero, is a point on a ratio scale that means the complete absence of an item being measured. For example, the zero point in temperature Kelvin has a special meaning in Physics, while the zero point in temperature Fahrenheit, or the zero point in Temperature Celsius (both of which are interval scales) does not mean the absence of heat.

> A ratio scale is an interval scale that has a non-arbitrary zero and allows only non-negative magnitudes.

Definition 2.6 — Ratio Scale

Multiplication and division can be performed on ratio scales in such a way that the operations are meaningful within the context of the domain. Unfortunately, unless this style of domain modeling approach is used, interval (and sometimes even ordinal and nominal) scales are usually implemented as integer or real numbers that permit multiplication by a positive constant. This multiplication is not correct with respect to the semantics of an application domain for any attribute scale type except ratio. As was discussed in the temperature example, multiplying a Fahrenheit temperature by two does not mean that the newest temperature is twice as hot as the old one. This point will be examined in context with the other scale types in the section on state transitions in this chapter.

The date scale is another example that illustrates differences between interval and ratio scales. Many date scales exist, but the one normally used in Western civilizations is based on the Gregorian calendar which is an interval scale with a non-absolute zero point. Days (which represent rotations of the earth) and years (which represent rotations of the earth about the sun) are two different units that are superimposed on the same scale. Since they are not even multiples of each other, complicated systems are used to convert between the units of the scale.

Because dates are attributes from an interval scale, it is not possible to make ratio comparisons between them. Much of the confusion that arises in computer implementations that deal with date and time occurs because of the lack of recognition that a date is a fundamentally different types of scales then an age or any other ratio attribute that measures quantity.

However, the difference between two interval scales results in a ratio scale. For example, age is an attribute typically derived from the leap year corrected difference between two dates. Since age is a ratio scale, it is possible to make proportional comparisons between the ages of two people. However, it would be impossible to make these same type of comparisons between two dates on a calendar.

2.3.9. Scale Summary

From a meta-classification level, the four scale types that have been discussed can be mapped on the following ordinal scale:

Nominal << Ordinal << Interval << Ratio where Y << X *means x is a scale of type y*

Scaling theory has been introduced as a well defined system for classifying value sets. Nominal scales categorize items and are the most basic scale type. Ordinal scales are nominal scales for which a strict ordering between categories can be maintained. Interval scales are ordinal scales with well defined units and a measurement granularity, and ratio scales are interval scales with a non arbitrary zero point. The relations between the different scale types are illustrated in Figure 2.3.

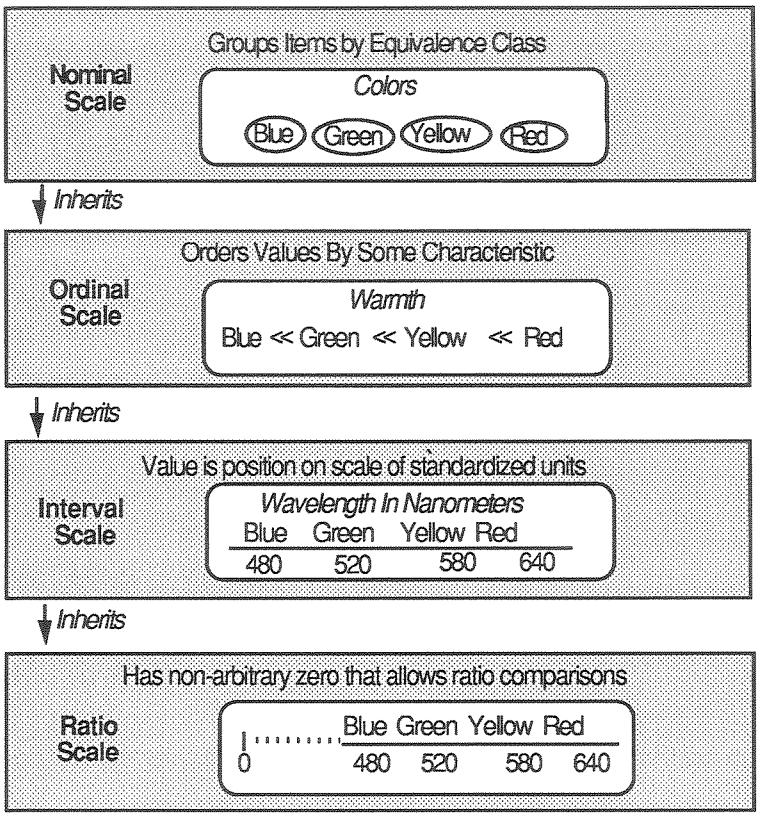


Figure 2.3 — Scales & Value Sets

2.4. Fundamental Units and Quantities

Units and scales have long played a critical role in knowledge representation. Six fundamental units are used as the primitives for construction of almost all models in the physical sciences. The fundamental quantities and units defined in the *International System of Units (SI)* are *Length in unit meters, Mass in unit kilograms, Time in unit seconds, Temperature in unit Kelvins, Electric current in unit amperes,* and *Luminous intensity in unit candelas*. Fundamental units are operationally, or procedurally, defined as illustrated in this definition of a meter:

> *The meter is a length that is exactly 1,650,763.73 times the wavelength of the orange light emitted when a gas consisting of the pure krypton nuclide of mass number 86 is excited in an electrical discharge, the wavelength being measured in a vacuum. [Shortley 73]*

In other domains, the same quantities exist, but different or additional quantities may be considered fundamental. For example, the quantity *Value in unit dollars* is a fundamental quantity within the class of United States business application domain models. In economic terms, a dollar is a value that was at one time directly related to the price of gold. However, the Gold Standard was eventually abandoned, and the actual value of the dollar now "floats". The United States government has operationally defined a dollar with the inscription found on the upper left portion of a one dollar bill:

Part of the task of modeling a domain is to identify the fundamental scales and units that are used across groups of domains. Beyond the scope of this paper are the philosophical issues involved in the belief that certain attributes are fundamental[5]. The meta-model recognizes these fundamental quantities and units so that they can be used as a basis for developing domain models. The quantities defined in the generalized meta-model are:

- length
- mass
- time
- temperature
- electric current
- luminous intensity
- value

2.4.1. Unit Conversions

Unit Conversions are another element of domain knowledge that are maintained within an attribute representation. Unit conversions implicitly represent the policies and procedures within different application domains. For example, the number of hours, and months within a year differs from industry to industry and represents policy decisions within any particular industry. In most industries, a work year consists of anywhere from 2,000 to 2,200 hours while the number of hours in a calendar year depends upon whether or not that year is a leap year. Furthermore, the number of months in a year depends on both accounting and policy decisions (e.g. an academic year is 9 months).

As was previously discussed, since calendar day and year are two distinctly different scales, it is difficult to make conversions between them. In most applications, unit conversions are performed only on ratio scales that measure quantities. The following predicates show some examples of unit conversions for the quantity time (These type of conversions are required for the unit_coercion discussed in Chapter 3:

```
unit_up(second, minute, 60),
unit_up(minute, hour, 60),
unit_up(hour, day, 24),
unit_up(day, week, 7),
unit_up(month, year, 12),
unit_up(year, decade, 10),
unit_up(decade, century, 10),

unit_dn(minute, second, 60),
unit_dn(hour, minute, 60),
unit_dn(day, hour, 24),
unit_dn(week, day, 7),

unit_dn(work_day,hours,8)
unit_dn(work_week, hours,40)
unit_dn(work_month, work_days, 20)
```

---

[5] [Wegner 88] discusses this from the standpoint of OOP. The philosophical discussion begins with Plato.

unit_dn(quarter, work_months, 3)
unit_dn(work_year, work_days,250)
unit_dn(year, days, 365)

2.4.2. Derived Quantities

Derived quantities are expressed in terms of *fundamental quantities*, and their units are specified in terms of the units that were used to create the particular quantity. In a domain model, these derived quantities are used to define attributes as well as being the basis for consistency and integrity checks. Recall that a necessary condition for the truth of any equation is that the units of each side be identical and that they make sense. The meta-model can go further and allow the domain expert to exclude derived quantities that will never make sense. For example, $/unit time, $/unit_length, $/ (unit_length)^2, $/ (unit_length)^3, are all reasonable unit combinations for an accounts receivables system but $/(unit_length)^4 and $\$^2$/unit_length are meaningless combinations of quantities.

While some may claim that derived quantities such as $/(unit_length)^4 could be meaningful in some systems, the modeling methodology rules out that particular derived quantity for all business applications. The line of reasoning follows.

The interpretation for $/(unit_length) is a derived quantity that is applied to buying rope, cable, or anything else which is measured by a linear dimension which is linear foot. A domain interpretation for $/unit_length^2 (which can also be expressed as $/unit_area) is a derived quantity that is used for expressing purchases of things measured in two dimensions such as rugs, sod, or wall paper. In the third case, things that are purchased by unit volume such as oxygen are measured in $/(unit_length)^3 (which can also be expressed as $/unit_volume). Although the preceding derived quantities have an interpretation within the physical world, there is no obvious extension to give meaning to the derived quantity $/(unit_length)^4. Figures 2.4 and 2.5 show screens to elicit these derived quantities and units.

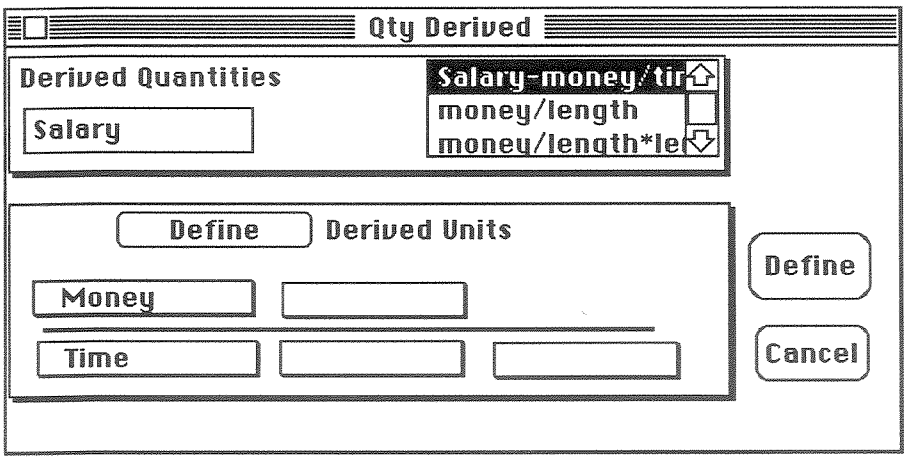


Figure 2.4 — Defining the derived quantity Salary (time/money)

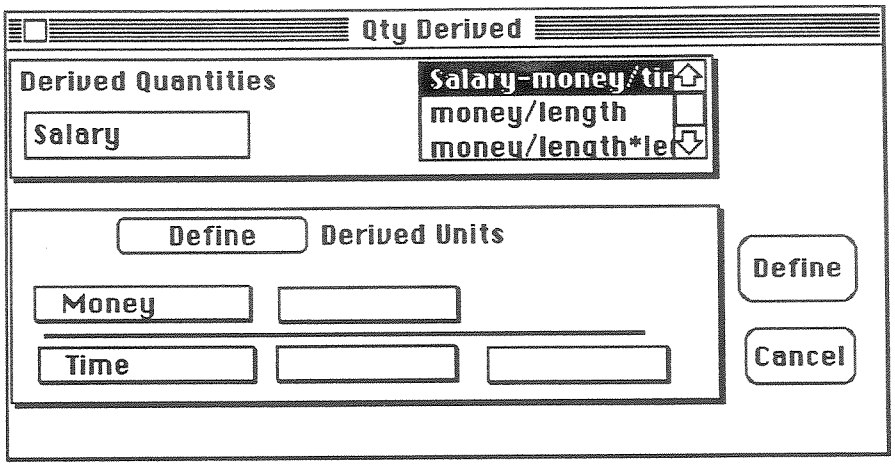


Figure 2.5 — Dollar/hour is one instance of salary

2.5. Population Parameters *PP*(A)

Associated with each attribute is a set of population parameters, *PP*(A), that describe the distribution of values within a value set. The characteristics of these distributions are used to create attribute subdivisions, determine input checks for the runtime system, determine default values, and infer missing values. While the values of the actual population parameters are always domain specific, the types of parameters are determined by the scale type as shown in Figure 2.6. The inheritance arrows in the figure show that while all scales have the statistical characteristics of nominal scales, additional characteristics are gained by scale type.

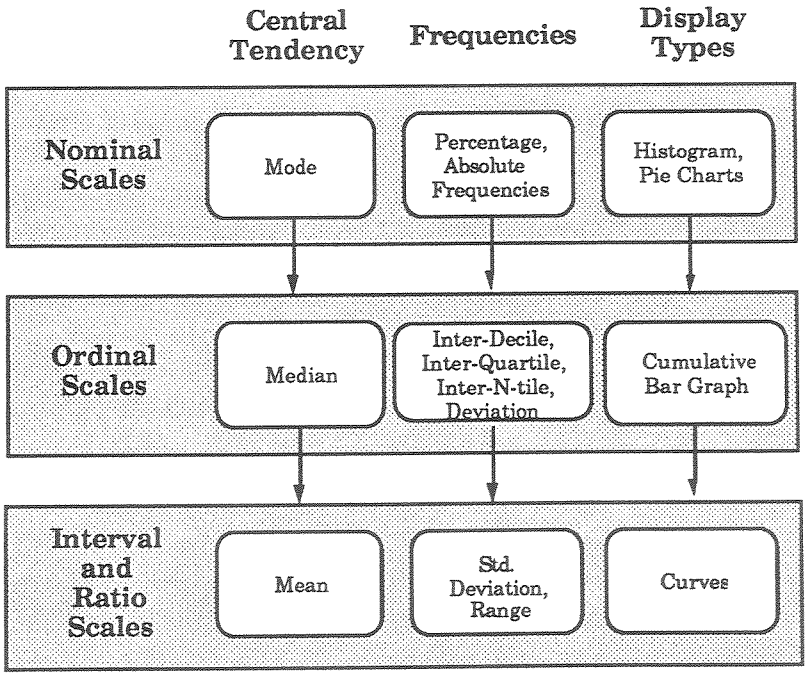


Figure 2.6 — Scale Type Determines Population Parameters and Displays

Even within a scale type, certain parameters are more appropriate than others. For example, the median is more appropriate than the mean to measure central tendency for a

highly skewed ratio-scaled attribute such as salary. The choice of the appropriate parameters is made by the domain expert but guided by standard statistical modeling principles (such as those described in [Blalock 72] and [Labovitz 72]). While anything but a cursory discussion of statistical techniques and probability distribution functions is beyond the scope of this paper, the next few examples are intended to convey the adaptability of well defined and accepted statistical techniques to domain modeling.

Population parameters are assigned by the domain expert, who is presumed to have knowledge of the overall population of an attribute value set within a specific domain. Throughout this dissertation, both the term population parameter and the term statistic are used. In general, a population parameter is assumed to be derived from complete knowledge of an entire population while a statistic is derived from a sample of that population.

Probability distribution functions are used to characterize value sets within physics, economics, manufacturing, operating systems, and a multitude of other fields. The next few sections briefly discuss a few of the most commonly occurring distribution functions.

2.5.1. Normal Distributions

A normal distribution is a smooth, symmetrical, bell-shaped curve that characterizes the distributions of many attribute value sets. Normal curves can be completely specified by a mean and a standard deviation. Because the normal curve is both symmetrical and unimodal, it's mean, median, and mode are all equal. A normal curve is shown in Figure 2.8, in the section on defaults.

2.5.2. Poisson Distribution

The Poisson Distribution is another commonly occurring distribution that is often used to represent a variety of time based phenomena such as the number of: radioactive particles emitted from a source, telephone calls arriving at a switchboard, or electrons emitted from a cathode ray tube during a fixed time, T. In order for a value set to be characterized by the Poisson Distribution, several assumptions must be made. The most important assumption is that the item count is independent of the particular interval of time, T. The poisson distribution is defined as follows:

$$e^{-\lambda t} (\lambda t)^{h} / (n^{i})$$

2.5.3. Exponential and Other Distributions

Another common distribution is the exponential distribution and its specializations the gamma and the chi-square distribution. The exponential distribution is defined as $f(x) = \alpha e^{-\alpha x}$ with the mean and variance defined as $\mu = 1/\alpha$ $\sigma^2 = 1/\alpha^2$.

Related to the exponential distribution is the gamma probability distribution which contains an extra parameter r. When the r parameter of the gamma distribution equals 1, the gamma distribution is equivalent to the exponential distribution. Finally, the chi-square distribution is a special case of the gamma distribution obtained when $r = n/2$ and $\alpha = 1/2$ and n is a positive integer.

While the distribution of value sets is often normal, it is not unusual to find value sets described by other types of distributions.

2.6. Initialization Procedure *I*(A)

The preceding sections of this chapter have introduced a method of classifying the value sets of an attribute. One of the benefits of this classification is that it may be used to accomplish tasks such as eliciting intelligent default information in a systematic well-organized manner without resorting to ad hoc methods or specialized code attachments.

Knowledge about the proper way to initialize an attribute is stored within a domain model as an initialization procedure. There are several issues to be considered when examining initialization. Default values, mandatory entry requirements which exclude null values, optional entry which allows null values, system defined values and user overridden values are some of the issues which must be considered for initialization of an attribute. Portions of the discussion concerning initialization of attributes must be deferred until the following chapter on classes, when context sensitive initialization of attributes is discussed. However, default values are an important part of a domain model, and the following section describes a methodology for their determination.

2.6.1. Defaults

Default values are system generated values intended *only* to assist users in filling in values, and not to make extended inferences about the information in a domain model. In Chapter 4, domain structuring using attributes and classes is further discussed, along with its relation to the artificial intelligence concept of defaults. While portions of the meta-model relate to that question, this section is concerned only with the generation of default values for the runtime enduser.

Some default values can be automatically generated by the runtime system through the use of $\mathcal{PP}$(A). Consider the nominally scaled attribute of a company defined with the following categories and proportions.

Company_type: Nominal_Scale
((Corporation .54) (Partnership .21) (Sole_ Proprietorship .23) (Other .02))

Because of the domain model's information about the relevant proportions, the runtime system can generate *corporation* the appropriate default using the algorithm shown in definition 2.6.

For a set of categories $\mathcal{C}(m) = \{C_1, \ldots C_n\}$ in a nominal scale $\mathcal{N}(m)$,
the default category C_i is the max(Proportion(C_i), . . . , Proportion(C_n)).
where Proportion(C_i) = $|C_i|/|\{C_1, \ldots C_n\}|$.

Definition 2.7 — Rule to Determine Default Category for a Nominal Scale

Similarly, the default value for an interval or ratio scale scaled attribute is the expected value of that attribute. It is possible that the expected value will not be a member of the value set. In this case, additional more sophisticated strategies can be used.

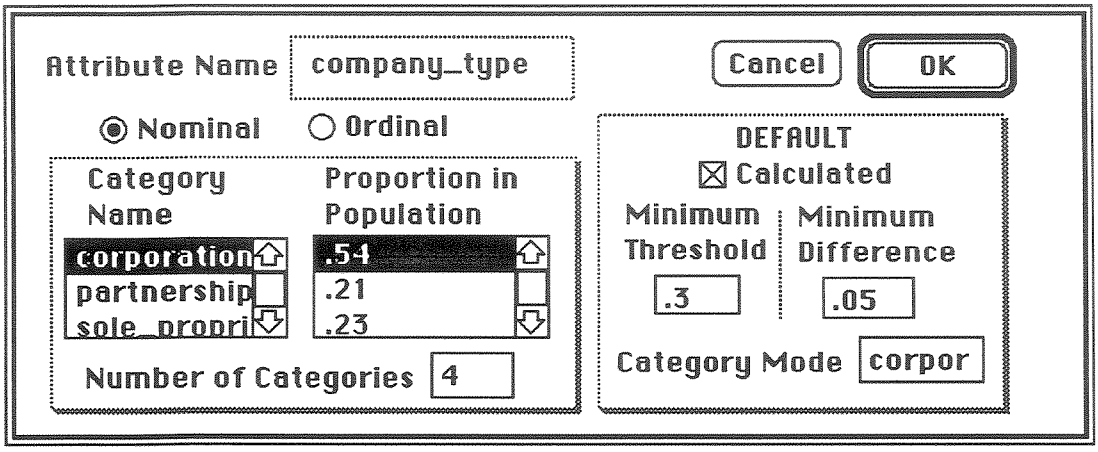


Figure 2.7 — Screen to elicit nominal/ordinal attribute categories

Any default selection scheme may be made more sophisticated by the addition of tuning parameters. In Figure 2.7, the system is directed to calculate a default using a *minimum threshold* and a *minimum difference*. In this case, a single default value will only be chosen if it meets the condition of being greater than the minimum threshold and the condition that it be at least the minimum difference or greater than the next ranked value.

The actual value of *Threshold_value* depends on a variety of factors that include Type I and Type II errors, as well as the expert's perception of risk. Elicitation of parameters of this type is discussed in [Keeney 76].

2.6.2. Multiple Defaults

Multiple defaults are a more sophisticated form of default value creation. Given that a domain model is able to generate default values, it can, when appropriate, generate a set of values from which the enduser can choose because the domain model can produce a set of defaults rather than just supplying an open ended question for the user, the system is able to reduce the enduser's task from an open ended question to a multiple choice selection. That can be displayed in rank order on a menu style display.

2.6.3. Displaying Information

By understanding scale type, the system is able to use predefined methods for displaying information for the end user. As was mentioned previously, categories of a nominal scale can be listed by probability in a menu system. Ordinal scales can also be listed in a menu system although they need to be ordered in the pattern corresponding to their ranking within the scale. Interval and ratio scales can be displayed in the form of thermometers, gauges, and other analog devices [Apple 85] as well as being displayed in ordinary digital format.

2.6.4. User Override of Default Values

Overriding default values is a different type of selection than allowing an initialized attribute to be changed. For example, in a military recruitment system, the gender category might be set to default to be male, but could be easily overridden to become female. However, once initialized it could not be changed if the probability of change of that

attribute is zero. This issue will be further discussed later in this chapter when $\mathcal{PC}(A)$ is introduced, and also in Chapter 4 when taxonomies are discussed.

2.6.5. Checking Input — Confidence Intervals

Population Parameters can also be used by the runtime system to check user input. Confidence interval checking is a more sophisticated form of ordinary range checking that matches runtime input values against the expected distribution of input. By using confidence intervals, an application designer can instruct a runtime system to suggest that the enduser be warned, with different levels of severity, about values outside of predefined confidence intervals.

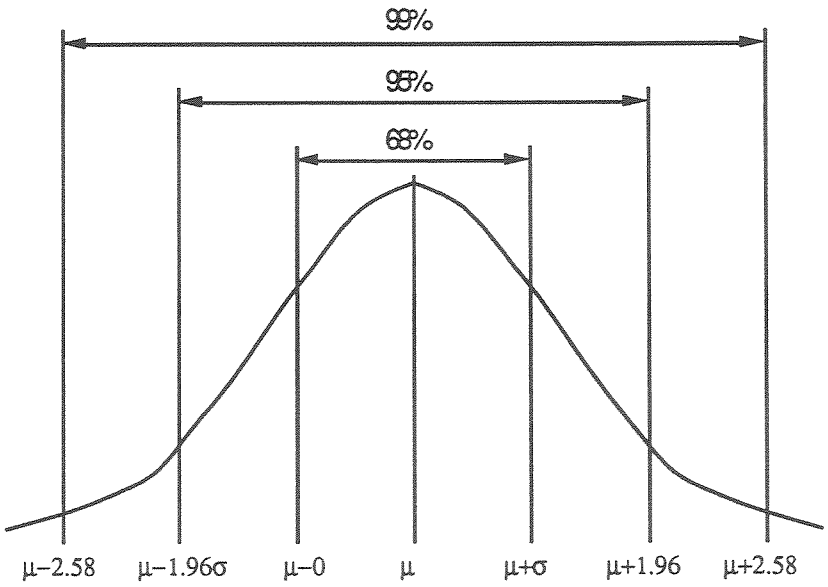


Figure 2.8 — Confidence Intervals Under A Normal Curve

For example, if the distribution of an attribute, A, can be assumed to be normal with mean μ, and standard deviation, σ, then the probability of A assuming any particular value, A.value can be bounded as follows:

$$\text{Prob}\,(\,|A_1.\text{value}| > \mu + 1.96\,\sigma\,) < 0.95$$

$$\text{Prob}\,(\,|A_1.\text{value}| > \mu + 2.58\sigma\,) < 0.99$$

Expressed in English, the last two statements tell us that 95% of the values will fall within 1.96 standard deviations from the mean and 99% of the values will fall within 2.58 standard deviations from the mean. Consequently the run time system has an algorithmic method of producing warning messages that resemble those produced by weaker methods. The following two statements give a specification for the run time system:

IF ($|A_1.value| > \mu+1.96\sigma$)
Then send_a_message(*Warning - only a 5% chance of this value*)

IF ($|A_1.value| > \mu+2.58\sigma$)
Then send_a_message(*WARNING - Only a 1% chance of this value*)

Obviously, the probability of an attribute attaining a value more than 3 standard deviations from the mean is exceedingly small.

2.6.6. Confidence Intervals — Unknown Distributions

The preceding discussion was predicated on the basis of a value set belonging to a normal distribution. In actual practice, many, but not all distributions can be modeled as normal distributions. Consequently, it is useful to be able to supply a similar, though less restrictive, system of confidence interval range checking for distributions that are not known to be normal. Chebyshev's theorem makes this possible. It states that the following rules apply for any distribution for which only the mean and standard deviation are known:

$$\forall x \; Prob(|A_1.value - \mu| >= x\sigma) <= x^{-2}$$

$$Prob(|A_1.value| > \mu+2\sigma) < 0.75$$

$$Prob(|A_1.value| > \mu+3\sigma) < 0.89$$

These rules allow the same type of confidence interval checking to be performed, and warning messages to be displayed that were discussed in the section on normal distributions.

2.7. Transitions Within Value Sets — Probability of Change $\mathcal{PC}$(A)

The probability of change of an attribute, $\mathcal{PC}$(A), is a value that ranges from 0 through 1 and indicates the likelihood that an attribute will ever change state within the lifetime of any particular domain model. One of the purposes of $\mathcal{PC}$(A) is to indicate the types of distinctions that exist between taxonomic classification devices such as animal_type(warm_blooded, cold_blooded) and sleep_activity(awake, asleep). In the first case, $\mathcal{PC}$(animal_type) = 0 because the state will never change. While in the second case $\mathcal{PC}$(sleep_activity) = 1 because the state will change on a regular basis.

The decision to allow changes is of course domain-specific. Some attributes may change, but not within the life of a particular domain model. Depending upon the role of an object within a model, certain attributes do not change once they are initialized. The birthdate of a person and the author of a book are examples of attributes that are invariant for the life of an object. Of course, there are many attributes whose meaning is clear and invariant within the model, but might be viewed differently outside of any particular domain. Consider the the gender attribute of a person object:

Person.Gender = Nominal_scale (Male, Female)

From a chromosome based definition of gender, this attribute can't be changed. However, if the domain model has some case specific use for knowledge of sex change operations, then this probability of change is incorrect. In the special case that $\mathcal{PC}$(A) = 0,

we say that A is an immutable attribute. Immutable attributes will be further discussed in Chapter 4 as the basis for taxonomic classifications.

A is an immutable attribute if $\mathcal{PC}(A) = 0$ $\quad \mathcal{PC}(A) = 0 \dashrightarrow \mathcal{R}(A) = \phi$

Definition 2.8 — Immutable Attributes

2.8. Measurement Scales $\mathcal{M}(A)$ and State Transition Relations $\mathcal{R}(A)$

In the preceding sections, we developed the notion of classifying attributes by their scale type and mentioned some of the potential benefits of such a system. By classifying an attribute in terms of its scale type, a domain model is able to make a number of assumptions concerning the characteristics of the attribute value set. In this section we expand on this idea and classify the state transitions in a way that helps the domain expert choose those transitions to be allowed.

In order to capture the semantics of an attribute, the meta-model helps the domain expert define a relation, $\mathcal{R}(A)$, on the value set, V, that specifies the possible transitions on the attribute. Earlier in this chapter, we developed the notion of scale type and showed how by classifying the scale type of an attribute, the meta-model can define generalized characteristics of the value set and population parameters for that attribute. In this section, we exploit the scale type of the attribute to aid the domain expert in selecting the transitions which take place on attribute value sets. This is possible because, when an attribute is classified by the domain expert as to scale type, the model then can present the domain expert a list of possible transitions to be included or excluded from an attribute definition. These transition categories become richer as an attribute moves from a nominal to a ratio scale.

2.8.1. Nominal scales

Given an attribute classified by a nominal scale that consists of n categories, there are n^2 possible transitions among those categories. Although n^2 possibilities exist, the semantics of the domain frequently exclude many, if not most, of these possible transitions. One way to approach the systematic exclusion of transition pairs is to ask the domain expert if identity transitions are allowed. In most cases an identity pair is a redundant mapping that comes about as the result of an incorrect assumption about program operation or a bad specification.

Identity $\quad (x, x) \in \mathcal{R}(A)$

Definition 2.9 — Identity Relation

2.8.2. Nominal Scale Example

The following example will make the preceding point clear. Consider a simple nominal scale, that represents a person's current marital status

marital_status = nominal_scale (never_married, married, divorced, widowed)

Without any domain specific semantic restrictions there are 4^2 or sixteen possible transitions among the values of this value set. The modeling technique formalizes these transitions in terms of a relation with sixteen pairs. The possible state transitions on the attribute marital status are represented by the relation

R = {(N, N), (N, M), (N, W), (N, D)

(M, N), (M, M), (M, W), (M, D)

(W, N), (W, M), (W, W), (W, D)

(D, N), (D, M), (D, W), (D, D)}

Obviously, within the context of Western culture, many restrictions are possible. We know that never_married can map to married, married to divorced or widowed; divorced to married; and widowed to married. The allowable transitions can be viewed pictorially in the following state diagram:

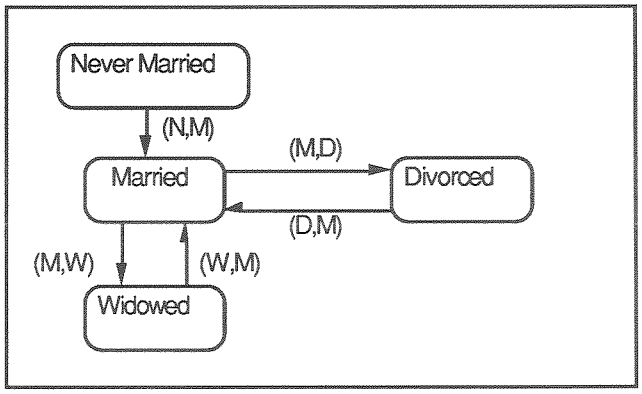


Figure 2.9 — State Transitions for Marital Status

As Figure 2.9 shows, domain semantics restricts the relation to the following allowable transitions:

R = {(N, M) (M, W) (M, D) (D, M) (W, M) }

In this simple example it was easy for a domain expert to manually enumerate the applicable pairs of the relation R. But with other scales, or with nominal scales that have more categories, the modeling methodology provides a more systematic methodology for the domain expert's selection of the appropriate subset of $\mathcal{U}(A) \times \mathcal{U}(A)$.

2.8.3. Ordinal Scales

For a nominally scaled attribute the identity subset is the only systematic way to exclude or include a subset of the state relation R. Ordinal scales, however, provide at least four more well defined subsets that can be easily excluded or included by a domain expert. Definition 2.10 shows a single step increment, a multistep increment, a single step decrement and a multistep decrement definition. The definitions define potentially allowable pairs for any ordinal scaled attribute. This means that if there is enough domain

knowledge to classify an attribute as to scale type, that scaling theory and the modeling methodology facilitate the easy elicitation of allowable pairs.

Single Step Increment	(x, y) --> $y \neq succ(x)$
Multi Step Increment	(x, y) --> $y \neq succ(x) \wedge x << y$
Single Step Decrement	(x, y) --> $y \neq pred(x)$
Multi Step Decrement	(x, y) --> $y \neq pred(x) \wedge y >> x$

Definition 2.10 — Additional Ordinal Scale Transition Pairs

As an example, consider education as an ordinal property of a person that could be broken into four groups: (No_high_school, high_school, college, graduate_school). The attribute education means "highest level of education obtained" which raises the classification of the educational scale from a nominal to an ordinal scale as shown below:

Education: Ordinal_Scale
(No_high_school << high_school << college << graduate_school)

We know that there are four possible values in the value set for this attribute. Realistically, the only transitions that we care about are the transitions which take the current state of highest level of education attained and change it to another higher state. Unrestricted state transitions result in n^2 possibilities. As was the case in the last example, there are 16 possible transitions.

But ordinal scales allow more systematic restrictions than a nominal scale. Many times the number of potential transitions can be restricted, causing the number of transitions to change from $O(n^2)$ to $O(n)$. One way to look at the n^2 possible cases is by summing the following relation subsets.

Identity	n possibilities
UpScale	$(n^2-n)/2$ possibilities
DownScale	(n2-n)/2 possibilities
Total	n^2

The selection of these subsets is determined by the meaning of the attribute within the context of the domain. Figure 2.10 shows a screen to gather this information.

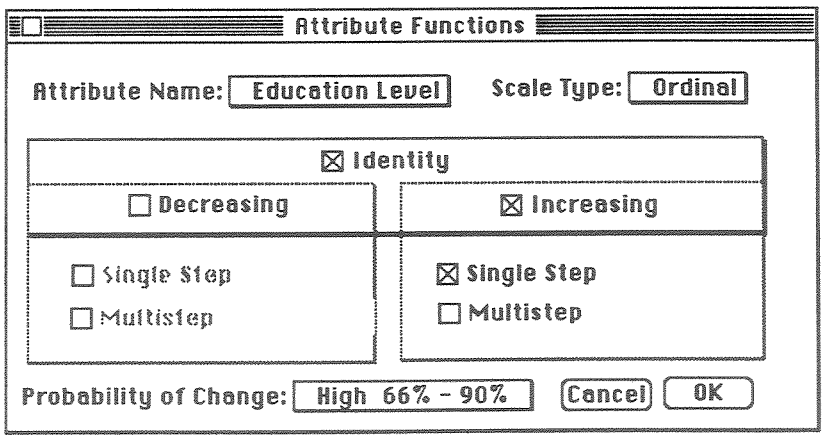


Figure 2.10 — Screen to elicit value set transitions

Let us assume that there are not going to be cases where degrees may be taken away. This would immediately eliminate all the decreasing transitions. Furthermore, assume that one must complete high school to attend college and must complete college to get a graduate degree. This reduces the general class of increasing transitions by only allowing single step transitions. In this example, there are only 3 increment transitions: graduation from high school, graduation from college, and completion of a graduate degree. We have excluded identity transitions for all but completion of a graduate degree.

Even with incomplete applications of domain knowledge, the number of potential transitions can often be reduced from order n^2 to order n. For example, in the general case, even if only multistep transformations are eliminated, n^2 -n possibilities are reduced to 2(n-1) selections as shown below:

UpScale Single Step	n-1 possibilities
DownScale Single Step	n-1 possibilities
Total	2n-2

2.8.4. Probabilities assigned to relations

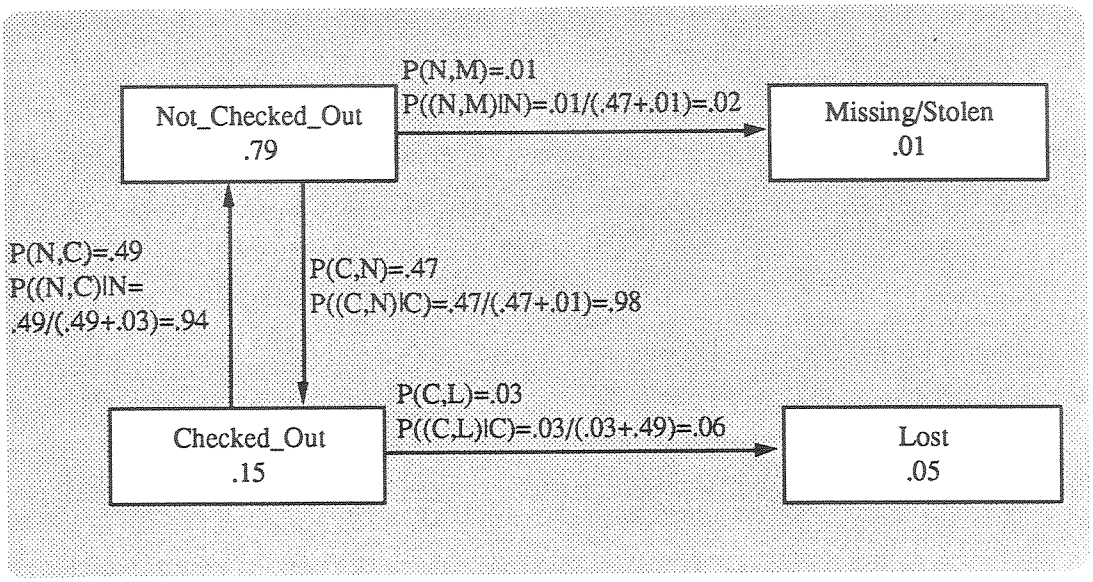
Assigning probability occurrences to each pair of the relation allows the system to order transitions for selection by the application designer (note that the section on defaults described how the system used probabilities to select defaults for endusers). In this manner it is not only a classification scheme, but also a system for prioritizing possible selections. Furthermore, this representation creates a simple way to accurately represent information that is usually stated in natural language or other informal terms. For example, the statement *"One fourth of our employees receive graduate degree during their term of employment"* can easily be represented in the model.

Probabilities are further illustrated with this example from the library problem. Consider the book check_out_status:

Check_out_status:

nominal scale (not_checked_out, checked_out, lost, missing_stolen)

Figure 2.11 shows state transitions and associated probabilities for this attribute. The modeling methodology captures three different but highly related probabilities of change that are associated with each attribute. Each of these numbers is supplied by a domain expert, but because of the relations between the probabilities, it is not necessary to elicit all of the numbers to complete the model.



PC(A) = .9

Figure 2.11 — Check-Out Status, State Transition Probabilities

The first set of probabilities to consider are the overall chances that any particular arc will be taken. From a high level perspective, the figure indicates that 47 percent of the transitions are book returns and three percent are book losses. However, if we know that a book has been checked out then the probability that it will be lost, given that it is checked out, is 6 percent, and the probability that it will be returned, given that it is checked out, is 94 percent.

At the bottom of figure 2.11, pc(a) = .9 shows the overall probability that some type of state change will ever take place in the attribute. Although for any particular book, the probability that it will be checked out, given that it hasn't been checked out, equals .98, pc(a) = .9 shows the overall chance of any transition taking place at any time for all books. This reflects the fact that it is not necessarily true that a state transition for a book will ever take place within the lifetime of a system (i.e., there is no guarantee of liveness).

2.8.5. Interval Scales

Because interval scales have a unit (As was presented in 2.3.7), addition and subtraction operations have semantic meaning for attributes of these scale types. Consequently, the following subsets of $\mathcal{R}$ can be used by the domain expert as a first cut at further restricting $\mathcal{R}$. In addition to the transitions defined for ordinal scales, the following transitions are applicable for an interval scaled value set V:

Unit addition	$(x, x + 1) \wedge (x \in V) \wedge (x + 1 \in V)$
Arbitrary increment	$(x, x + i) \wedge (x \in V) \wedge (x + i \in V) \wedge (i > 0)$
Unit subtraction	$(x, x - 1) \wedge (x \in V) \wedge (x - 1 \in V)$
Arbitrary decrement	$(x, x - i) \wedge (i \in V) \wedge (x - i \in V) \wedge (i > 0)$

Definition 2.11 — Relation Subsets of Interest in an Interval Scale

2.8.6. *Ratio Scales*

Ratio scales give domain-specific meaning to multiplication and division operations. Since ratio scales are interval scales, they inherit the relation subsets defined in definition 2.11 and add the following candidate relation subsets to the selection supplied to the domain expert for exclusion or inclusion within a domain model:

Multiplicative increase	$(x, x * i) \wedge (x \in V) \wedge (x * i) \in V) \wedge (i > 0)$
Multiplicative decrease	$(x, x * i) \wedge (x \in V) \wedge (x * i) \in V) \wedge (0 < i < 1)$

Definition 2.12 — Additional Subsets of Interest in a Ratio Scale

2.8.7. *Summary of Transition Relations*

Section 2.3 showed the relationship between scale types and value sets. We saw that an ordinal scale adds additional information to a nominal scale which allows further characterization of a value set. After units were introduced, it was shown that an interval scale is a specialization of an ordinal scale and that a ratio scale is an interval scale with a non-arbitrary zero. Figure 2.3 graphically summarized these results.

Definitions 2.9, 2.10, 2.11, and 2.12 defined potential subsets of R that can be easily included or excluded by the domain expert. Given that an attribute is classified by its scale type, the modeling methodology provides an efficient and systematic method for further characterization of attribute state transitions. Figure 2.12 shows state transitions characterized in terms of the four basic scale types. Nominal scales are such simple types that the only systematic way to approach the classification of their state transitions is to consider the identity state transitions.

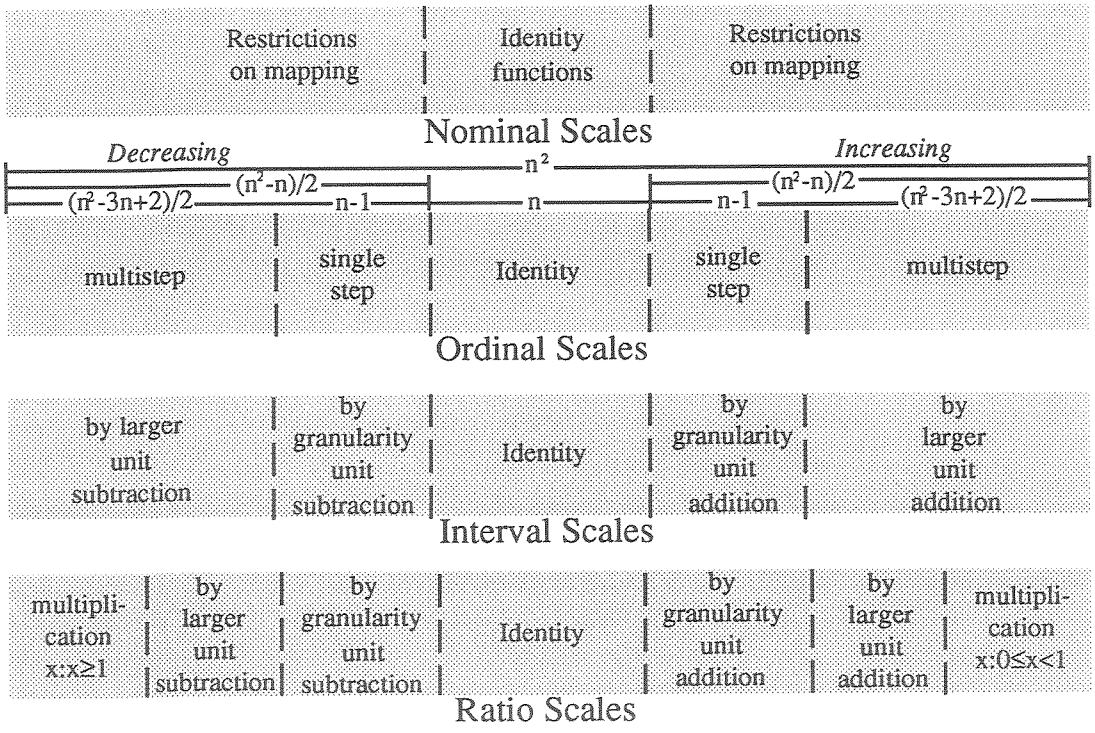


Figure 2.12 — Value Set Transitions

2.9. Axiom Summary χ(A)

In a meta-modeling approach such as the one described here, axioms are traditionally used to place additional restrictions on value sets; or on the operations performed upon those value sets. Axioms are also used to codify statements about the nature of particular attributes. In other models or specification languages they facilitate additional storage of domain knowledge by providing a well defined format in which to store information about an attribute.

Within this modeling methodology, axioms are defined in terms of the attribute definition. A statement that might normally be considered a classic axiom such as: $(x : 18 < x < 65)$ is built directly into the model. While in some systems this statement might be a constraining axiom listing the minimum and maximum ages of new employees, in this model this type of information is stored as a population parameter — in this case the minimum and maximum of a particular attribute.

After derived attributes are defined in Chapter 4, a series of axioms that act on attributes will be defined. However, the meta-model formalization of an attribute is sufficient to describe almost all of the restrictions that would be present at the attribute level. Within this meta-model a formalized system exists that captures the information normally specified by axioms. There are, however, a few significant omissions.

Depending upon the complexity of a part number, a security code, or other sometimes obscure types of input, additional restrictions are needed at the detailed input level. In previous work [Iscoe 86] I described in detail the lower level attribute checking features of a commercial program generation system called CRT Form. Some additional constraints included in that system but not specified in this dissertation meta-model include display features such as justification (left, center, right), character conversion (lower to uppercase, first letter uppercase and so on), number of digits displayed for a number including number of digits to the right of the decimal point, bit maps of actual characters allowed in text field and other associated details. These low level and visually oriented features, while important in actual commercial systems, are details that could easily be added to this meta-model but are not relevant to this dissertation.

2.10. Transformational Code Generation Issues

Although the primary subject of this dissertation is the semantic modeling of domain concepts, the representation can be easily used to address a number of implementation issues. These include data storage, elicitation of boundary information, and the exploration of alternatives which is sometimes referred to as "what if" analysis.

2.10.1. Data Storage

Understanding the range of an attribute allows decisions to be made concerning data allocation and storage. A nominal scale is a set of items that can be implemented as a byte. If the definition occurs on a byte boundary, the system can probe for likelihood of change. For example, if 50 states are defined in a mail order system, then one byte will certainly be sufficient for the life of the system, however, if a single byte is used to represent 200 city names, it may be courting disaster for a growing mailorder firm.

As a further example, if an attribute classified on an interval scale ranges between 1 and 200, and has a granularity of 1, it can be stored in a single byte with room left over for a property based on a 56 category nominal scale.

2.10.2. Elicitation of boundary information

In the last example, if the upper limit of a property had a good chance of being 400, then it would have been foolish to allocate only one byte of storage. Understanding the potential spread allows the system to elicit more information around sensitive range boundaries such as 255 or 32,767.

2.10.3. Exploration of alternatives--"What if?" analysis

Users rarely realize the full impact of a particular requirement or specification. Formalized scales provide the basis for having a system that can allow the user to interactively explore the effect of specification changes on an application program. Some of the tradeoffs normally performed by system analysts can be presented to the enduser in a form that they can understand.

For example, a common design decision is the tradeoff between potential future system expansion and current system efficiency and storage capability. System analysts are well versed in this type of decision making, and can phrase questions in an appropriate manner. However, application designers not trained in computer science would probably not realize that their request to make an attribute range from 1 to 1,000 would in general give them the flexibility of expanding that range to 65,000 while setting an initial range at 200 might limit their expansion to 256 or 512. Furthermore, an automated system can point out the future effects and cost (in terms of data storage and possible retrieval slowdown) tradeoffs inherent in a particular specification. Because it is straight forward for a computer system to understand these type of simple data level mappings, it is possible to set up an interactive system that can allow the enduser to ask *what if* questions and give the user tradeoffs that would ordinarily be provided by a human.

2.11. Attribute Summary

Throughout this chapter, we have described a systematic methodology for helping the domain expert specify constraints and restrictions on attribute value sets and transitions on those value sets. The methodology begins by borrowing from scaling theory the formalisms that have been developed for capturing measurements of properties. For the more quantitative interval and ratio scales, fundamental and derived units and quantities were introduced as a way of recognizing standard descriptions that occur in the physical world. Granularity was introduced as a further specification attached to a unit.

From statistics, we have used the notion of population parameters to characterize the distributions of the attribute value sets. Initialization procedures were defined as a way to handle defaults, multiple defaults, and introduce new methods of input checking.

After value sets were defined and characterized, the notion of a transition within the value set was introduced in terms of the modeling methodology. By classifying an attribute as to scale type, the domain expert is more easily able to specify the relevant transition pairs. Finally, attribute restrictions were introduced as a way to further subdivide attribute libraries.

This chapter has specified the meta-model's characterization of an attribute. In the next chapter, classes, which are collections of attributes, will be introduced. Their introduction will allow the definition of derived attributes and summary statistics on attributes.

Chapter 3 – Classes

3.1 Introduction to Classes

Classes encapsulate sets of attributes, provide for the definition of derived attributes, allow additional operations to be defined, are responsible for object instantiation and deletion, allow axioms to be defined, and maintain summary statistics about instantiated objects. The previous chapter described the primitive attributes from which classes are constructed. This chapter introduces classes and describes how the meta-model further captures domain semantics.

Domain experts use classes to organize information within application domains. As a practical matter, it is often easier to elicit attributes by concentrating on individual classes than it is to define attributes without the context of a particular class. By creating attribute and class libraries that can be reused throughout application domain models, a domain expert can build a set of basic classes that can be used to organize and structure information within a domain. Chapter 4 explains hierarchical decomposition and how it is used to specialize classes into sub-classes. Chapter 5 defines composition, the basic constructor of the modeling methodology and its use in creating larger classes from smaller ones. Finally, Chapter 6 defines association and its use in establishing operational relationships between classes.

3.1.1 Class Definition

Definition 3.1 begins the definition of a class. Sections 3.2 through 3.7 expand upon and elaborate this definition.

A class C is:	
• a set of attributes $\mathcal{A}$(C) and their subsets,	{Section 3.2-3.6}
Primitive Attributes $\mathcal{P}$(C),	
Derived Attributes $\mathcal{D}$(C),	
Naming Attributes $\mathcal{N}$(C),	
Referential Attributes $\mathcal{R}$(C),	
• a set of functions $\mathcal{F}$(C) that are used to create $\mathcal{D}$(C),	
These functions are expressed in terms of attributes.	{Section 3.2}
• a set of Operations as Follows:	{Section 3.5}
Object Instantiation $\mathcal{I}$(C),	
Object Removal $\mathcal{R}$(C),	
Other Operations $\mathcal{O}$(C),	
• a statistical summary function $\mathcal{S}$(C),	{Section 3.6}
• a set of axioms or integrity constraints $\mathcal{X}$(C)	{Section 3.7}

Definition 3.1 — Class

3.1.2 Object Definition

Definitions of objects[1] differ in many areas, but most share at least the following definitional fragment: *An object, in an object-oriented model, is an abstraction of an entity that is characterized by its state and a set of operations that access or change that state. State is defined in terms of the properties or attributes of an object.*

In this dissertation, the detailed definition of attributes gives a more detailed meaning to the term "state". Within the meta-modeling technique, objects are defined as follows:

> Given a class C with attributes $A_0, A_1, \ldots A_n$, an object O of Class C is defined to be a family of vectors of the form $[a_0, a_1, \ldots a_n]$ where $a_j \in \mathcal{V}(A_j)$ and where the vectors have a common value (the name of O) on the naming attributes of C.

Definition 3.2 — Object

Distinctions between classes and objects vary widely among different object-oriented systems. In spirit, the class/object distinction defined by the meta-model in this dissertation is most closely related to Eiffel [Meyer 88]. Classes serve as type-templates for objects and can be regarded as compile-time constructs that define how runtime objects are to be created. However, the class construct also maintains runtime information such as summary statistics about the objects created by a class. Although classes serve as both compile-time type definitions and repositories for runtime information, the distinctions between objects and classes is carefully maintained within a domain model. Classes are not objects and are never instantiated from other classes. Figure 3.1 further illustrates attributes, classes and instantiated objects Chapter 4 will explain the distinctions between classes, subclasses, and superclasses and give definitions for how subclasses are created from the attributes of superclasses.

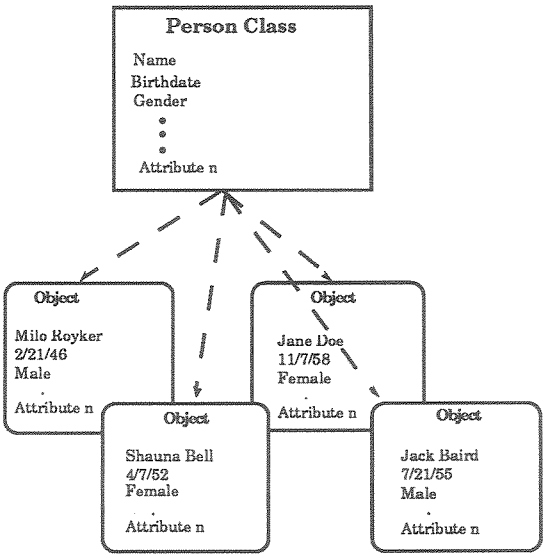


Figure 3.1 — Object Instantiations

[1] Some of the more commonly cited are [Dahl 66], [Goldberg 83], [Jackson 83], [Stefik 85], [Cox 86], and [Wegner 88]. Many other definitions exist. Chapter 7 contains a more detailed discussion of the term object-oriented.

3.2 Attributes - Primitive $\mathcal{P}(C)$ and Derived $\mathcal{D}(C)$

By using domain knowledge, attributes are partitioned into two groups within a class. *Primitive attributes* are considered to be the fundamental or basic properties of a class, while *derived attributes* are computed from primitive attributes and global knowledge (encoded as system functions) such as current_date.

The characterization of an attribute as primitive or derived is a function of the application domain and the goals of the domain expert. For example, a rectangle with the attributes *length*, *width*, *perimeter*, and *area* can be viewed in different ways. For most purposes, the attributes *length* and *width* are the primitive attributes from which the attributes *area* and *perimeter* are derived. However, a domain-specific decision to view other attribute pairs (such as *length* and *area*) as primitive attributes is reasonable and depends on the view of a class in the context of a particular application domain. Definition 3.3 defines primitive and derived attributes as well as introducing the functions $\mathcal{F}(C)$ required to create derived attributes.

The set of attributes $\mathcal{A}(C)$, can be viewed as two disjoint subsets: The fundamental, or primitive, attributes $\mathcal{P}(C)$ and the derived attributes $\mathcal{D}(C)$;

$$\mathcal{A}(C) = \mathcal{P}(C) \cup \mathcal{D}(C) \text{ and } \mathcal{P}(C) \cap \mathcal{D}(C) = \phi.$$

The $\mathcal{D}(C)$ require the introduction of a set functions $\mathcal{F}(C)) = \{f_j\}$, such that if $\mathcal{P}(C) = \{P_0, P_1, \ldots P_n\}$, $\mathcal{D}(C) = \{D_0, D_1, \ldots D_m\}$, and S is the special class of system attributes (such as current date), and $\mathcal{DOM}(S)$ is the cross product of the value sets of the primitive attributes of S, then there is a function $f_j \in \mathcal{F}(C)$ such that f_j is an *onto* function and $f_j : \mathcal{V}(P_0) X \ldots X \mathcal{V}(P_n) X \mathcal{DOM}(S) \rightarrow \mathcal{V}(D_i)$.

Definition 3.3 — Class (Primitive and Derived Attributes)

3.2.1 *Derived Attributes $\mathcal{D}(C)$*

The class definition introduces operations, $\mathcal{F}(C)$, which compute the values of derived attributes. Examples of an $\mathcal{F}(C)$ function are the algorithm to compute the age of a person. If objects of type person will persist in an application program for more than a year, then their age will change and will have to be derived from the leap year adjusted difference between a person's birthdate and the system supplied current date.

This is a domain-specific decision. If one were creating a computer dating service, it might be better to store age (and therefore it would have to be primitive) rather than to derive it. The decision to make age a derived (as opposed to a primitive) attribute is based on the assumption that an object's age will change within the life of an application.

Another example of an $\mathcal{F}(C)$ is the following cost_replacement function for a library book.

```
book.cost_replacement :=
    (plus (plus (times book.weight shipping_cost_per_pound) (book. cost)) (book.overhead))
```

In this example, the derived attribute book.cost_replacement is calculated by a formula that includes primitive attributes representing book weight and cost, a class shipping cost-per-pound and a factor which is included for overhead.

The functions $\mathcal{F}(C)$ sometimes requires system attributes, like system date, that return values used for calculations. Additional system attributes include standard items such as system time, and events that indicate user activity such as entering or updating information about a class. These system attributes were referenced in the class definition as $\mathcal{DOM}(S) \rightarrow \mathcal{V}(D_i)$. The system events will be further discussed in the section on class axioms. Finally, it should be noted that derived attributes can be defined in terms of other derived attributes in a recursive fashion where the recursion is guaranteed to terminate.

3.2.2 Arithmetic Functions on Attributes

Attributes of compatible scale types can be added or subtracted, multiplied or divided by each other and by constants. The next few sections explain addition and subtraction in detail. These algorithms will be used again in Chapter 5 when class composition is discussed.

3.2.3 Semantics of (Plus A1 A2) -> A3

Adding two attributes together to create a third attribute is a conceptually simple operation that in actual practice requires both domain knowledge and general rules concerning attributes. For example, it does not make sense to add time and temperature together, nor to add two dates together to create a third date. Humans intuitively understand that attributes of different quantities cannot be added together. The modeling methodology uses well-known techniques and results from statistics and basic measurement theory to define arithmetic operations between attributes.

One general domain-independent rule based on measurement granularity that helps determine how measures can be combined is that: *neither original nor derived information should ever be expressed at a finer level of granularity than is justified by its component parts, the algorithms used to compute it, or by its original level of measurement granularity.* Within the context of physical experiments, this concept is expressed by the rule that final results must not be expressed to a greater number of significant figures than the number of significant figures of the least precise measurement.

Domain knowledge and scaling theory tell us that the weight of an automobile built from only an Engine and a Chassis will be the sum of the weight of the engine and the weight of the chassis. Within the context of a particular attribute, such a rule might read, "*When you add two weight attributes, you must use a unit at least as coarse as the coarsest unit of the classes, convert all scales to that unit, and then add weight amounts to produce a new weight for the composite.*"

An informal rule that allows us to determine how to combine two ratio scale attributes is shown in figure 3.2:

In order to add or subtract two attributes with ratio scales, say A_1 and A_2, into a new attribute A_3 :

1) A_1, A_2, and A_3 all must be of the same quantity or derived quantity
2) A_1, A_2, and A_3 all must have the same unit measure or derived unit measure
3) The measurement granularity of the unit of A_3 must be at least as coarse as the most coarse of the measurement granularities of the units of A_1 and A_2.

Figure 3.2 — Necessary Conditions for Adding Ratio Scaled Attributes

The following algorithm for the attribute semantic, (Plus A_1 + A_2) -> A_3, details the process of adding attributes A1 and A2 to create a new attribute A3. Following the algorithm a further explanation by line number is given.

(Plus A_1 + A_2) -> A_3

```
1.  Procedure Add_attribute(A1,A2:Attribute;VAR A3:Attribute)
2.  Begin
3.  IF (A1.Scale = RATIO AND A2.Scale = RATIO)
4.     AND (A1.quantity = A2.quantity) AND COMPARABLE (A1.T,
                A2.T)
5.  Then Begin
6.        If  (A1.unit = A2.unit)
7.        Then
8.                 If  (A1.gran = A2.gran)
9.                 Then Begin
10.                              A3.Scale := RATIO;
11.                              A3.quantity := A2.quantity;
12.                              A3.unit := A2.unit;
13.                              A3.gran = A2.gran;
14.                              A3.T  := A1.T;
15.                              A3.min := A1.min + A2.min;
16.                              A3.max := A1.max + A2.max;
17.                              build_scale(A3.min,A3.max,
                                                    A3.gran,A3.V);
18.                              A3.EV := A1.EV  + A2.EV;
19.                              A3.SD2 := A1.SD2  + A2.SD2
                                 {assumes independence}
20.                              End
21.                    Else Begin
22.                              coerce_gran(A1,A2, Aprime1, Aprime2);
23.                              Add_attribute(Aprime1, Aprime2, A3);
24.                              End
25.                Else Begin
26.                        coerce_unit(A1, A2, Aprime1, Aprime2, error);
27.                        add_attribute(Aprime1, Aprime2, A3);
28.                        End
29.       Else ERROR( "Mismatched Scales")
30.       End; {add_attribute}

31.       Procedure coerce_unit(A1,A2:Attribute; VAR Aprime1,
                                                    Aprime2:Attribute,
                VAR error:boolean);
```

```
32.   Begin
33.         If bigger_unit(A1.unit, A2.unit)
34.         then begin
35.               new_att_new_unit( A2, A1.unit, A2prime)
36.               A1prime := A1;
37.               end
38.         else begin
39.               new_att_new_unit( A1, A2.unit, A1prime);
40.               A2prime := A2;
41.               end
42.   End; {coerce unit)

43.   Procedure new_att_new_unit(oldatt:Attribute; newunit:TyUnit;
            newatt:Attribute, VAR error:boolean);
44.   Begin
45.         newatt.T  := oldatt.T;
46.         newatt.Scale := oldatt.Scale;
47.         newatt.quantity := old_att.quantity;
48.         newatt.unit := old_att.newunit;
49.         unitfactor( oldatt.unit, newunit, factor);
50.         newatt.gran = old_att.gran/factor;
51.         newatt.min := oldatt.min * newatt.gran;
52.         newatt.max := oldatt.max * newatt.gran;
53.         build_scale(newatt.min, newatt.max, newatt.gran,
                                                      newatt.V);
54.         newatt.EV := old_att.EV * newatt.gran;
55.         newatt.SD2 := old_att.SD2 * newatt.gran**2;
56.   End; {new_att_new_unit}

57.   Procedure coerce_gran(A1,A2:Attribute; VAR Aprime1,
                                    Aprime2:Attribute);
58.   Begin
59.         If bigger_gran(A1.gran, A2.gran)
60.         then begin
61.               new_att_new_gran( A2, A1.gran, A2prime)
62.               A1prime := A1;
63.               end
64.         else begin
65.               new_att_new_gran( A1, A2.gran, A1prime)
66.               A2prime := A2; end
67.   End; {coerce gran}

68.   Procedure new_att_new_gran(oldatt:Attribute;
                  newgran:TyUnit; newatt:Attribute);
69.   Begin
70.         newatt.T  := oldatt.T;
71.         newatt.Scale := oldatt.Scale;
72.         newatt.quantity := old_att.quantity;
73.         newatt.unit := old_att.unit;
74.         newatt.gran = newgran;
75.         factor := newgran/oldatt.gran;
76.         newatt.min := oldatt.min * factor;
79.         newatt.max := oldatt.max * factor;
```

```
80.          build_scale(newatt.min, newatt.max, newatt.gran,
                                                    newatt.V);
81.          newatt.EV := old_att.EV * factor;
82.          newatt.SD2 := old_att.SD2 * factor**2;
83.    End; {new_att_new_gran}

84.    Procedure unit_factor(oldunit, newunit, VAR unitfactor,
                                    VAR error:boolean);
85.    Begin
86.          Query(oldunit, newunit, unitfactor);
87.    End; {unit_factor)

88.    Procedure build_scale(A3.min, A3.max, A3.gran; VAR A3.V);
89.    Begin
90.          value = A3.min
91.          A3.V = Null
92.          Repeat
93.                A3.V := Union(A3.V, value)
94.                value := value + A3.gran
95.          Until value > A3.max
96.    End; {build_scale)
```

Figure 3.3 — Attribute Semantics : (Plus A_1 A2) -> A3

Statement 1-5 Validity of Addition

Statement one is the procedure declaration to add attribute A1 and A2 to create attribute A3. Statement three requires that both of the attributes be ratio scales. As was explained in Chapter 2, neither nominal nor ordinal scales allow addition operations. Although interval scales allow addition operations, the addition of interval scales to create a new attribute has no meaning within most application domains.

Statement 4 states that, for two ratio scales to be added together they must, at a minimum, be of the same quantity and allow the same transitions on their value sets. The quantity check eliminates obvious semantic problems such as adding time plus temperature. The restriction on transitions prevents mismatched scales from being combined.

Unit Coercion Statement 6, 25-28, 31-56

Unit coercion is required in the event that the units of the two attributes do not agree. Consider the addition of two attributes defined as follows:

A1: Time in Hours (granularity .25)

A2: Time in minutes (granularity 1)

Obviously, in order to combine the two attributes, A2 will have to be coerced from minutes into hours. Statements 31 to 42 determine which attribute needs to be changed, after this determination, the procedure new_att_new_unit (line 43) is invoked. Unit factor (line 84) retrieves a factor that accounts for the magnitude of difference between the two units. Unit factor was first described in Chapter 2, in the section on unit conversions.

After the factor is defined, a new granularity is assigned (line 50) a new minimum and maximum are defined and a new scale is constructed (line 53). Finally, a new expected value and standard deviation are computed. The result of the coercion is:

A1: Time in Hours (granularity .25)
A2: Time in Hours (granularity 1/60)

Statement 8, 21-24, 57-67: Granularity Coercion

After the units are adjusted, granularities must be made the same which is done using the rule previously described. After granularity coercion, both attribute A1 and A2 can be combined:

A1: Time in Hours (granularity .25)
A2: Time in Hours (granularity .25)

Statement 10-14: Quantity, units, granularity

These statements make the assignment of a scale type, quantity, unit, and granularity into a new attribute.

Statement 15-17: Value Set Creation

A value set is created from the new attribute in the following way. First the minimum and the maximum value for the new attribute are defined. Then, since the value set of an attribute is defined in terms of its multiples of unit granularity, a new attribute is created. Build_scale constructs a value set for scale from its minimum, maximum and unit granularity.

Statement 18: Expected Value

The expected value of the final attribute is the sum of the expected values of the initial attributes A1 and A2. Because attributes are characterized, by at least, their mean and standard deviation, we are able to use well-studied properties for computing the population parameters. For example, it is straight forward to prove that $E(x+y) = E(x) + E(y)$.[2]

Let (X,Y) be a two-dimensional random variable with a joint probability distribution. let Z = $H_1(X,Y)$ and W = $H_2(X,Y)$.

$$E(Z+W) = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} [H_1(x, y) + H_2(x, y)]f(x, y)\, dx\, dy$$

where f is the joint pdf of (X, Y)

$$= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} H_1(x, y)\, f(x, y)\, dx\, dy + \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} H_2(x, y)\, f(x, y)\, dx\, dy$$

$$= E(Z) + E(W)$$

Let X and Y be any two random variables, and Let $H_1(X,Y) = X$, and $H_2(X,Y) = Y$. then $E(X+Y) = E(X) + E(Y)$.

Figure 3.4 — Proof E(x+y) = E(x) + E(y)

---

2 Proofs are adapted from [Brownlee 65], [Feller 50], [Feller 71]

Statement 19: Variance

In a similar manner we can show that, assuming that A1 and A2 are independent variables, then V(A1+A2) = V(A1) + V(A2). The independence assumption means that certain combinations of derived attributes are not able to be combined in this manner.

$$V(X) = E(X^2) - [E(X)]^2$$

$$V(A1 + A2) = E(A1 + A2)^2 - (E(A1 + A2))^2$$

$$= E(A1^2 + 2A1A2 + A2^2) - (E(A1)^2 - 2E(A1)E(A2) - (E(A2))^2$$

$$= E(A1^2) - (E(A1))^2 + E(A2^2) - (E(A2))^2$$

$$V(A1 + A2) = V(A1) + V(A2)$$

Figure 3.5 – Proof V(A1+A2) = V(A1) + V(A2)

3.2.4 *Semantics of (Plus A_1 C) -> A3*

While two interval scaled attributes are rarely added together, it is common to add either a ratio or an interval scaled attribute to a constant as shown below:

(Plus A_1 C) -> A_3

```
1.  Procedure Add_attribute_plus_C(A1,C:Attribute;VAR A3:Attribute)
2.  Begin
3.  IF ((A1.Scale = RATIO AND C.Scale = RATIO)  OR
        (A1.Scale = INTERVAL AND C.Scale = INTERVAL))
4.      AND (A1.quantity = C.quantity)
5.  Then Begin
6.      If  (A1.unit = C.unit)
7.      Then
8.              If  (A1.gran = C.gran)
9.              Then Begin
11.                             A3.quantity := A1.quantity;
12.                             A3.unit := A1.unit;
13.                             A3.gran = A1.gran;
14.                             A3.T  := A1.T;
15.                             A3.min := A1.min + C;
16.                             A3.max := A1.max + C;
17.                             build_scale(A3.min,A3.max,
                                                A3.gran,A3.V);
18.                             A3.EV := A1.EV  + C;
19.                             A3.SD2 := A1.SD2;
                                        {assumes independence}
20.                             End
21.                     Else Begin
22.                             coerce_gran(A1,C, Aprime1, Aprime2);
23.                             Add_attribute(Aprime1, Aprime2, A3);
24.                             End
25.             Else Begin
26.                     coerce_unit(A1, C, Aprime1, Aprime2, error);
27.                     add_attribute(Aprime1, Aprime2, A3);
28.                     End
29.     Else ERROR( "Mismatched Scales")
30.     End; {add_attribute_plus_C}
```

Figure 3.6 — Attribute Semantics : (Plus A_1 C) -> A3

(Plus A_1 + C) -> A_3 resembles the algorithm for the addition of two attributes, the only difference can be found in statements 3, 18 and 19. The proof for Statement 19 is as follows:

$$
\begin{aligned}
V(X + C) &= E[(X + C) - E(X + C)]^2 \\
&= E[(X + C) - E(X) - C)]^2 \\
&= E[X - E(X)]^2 \\
&= V(X)
\end{aligned}
$$

Figure 3.7 – Proof V(X+C) = V(X)

3.2.5 Semantics of (Times A_1 C) -> A_3

Multiplication by a constant is valid for interval scales in the case where the scale is being converted from one unit to another as described below.

(Times A_1 C) -> A_3

```
1.  Procedure Times_Attribute_C(A1,C:Attribute;VAR A3:Attribute)
2.  Begin
3.  IF ((A1.Scale = RATIO AND C.Scale = RATIO)  OR
        (A1.Scale = INTERVAL AND C.Scale = INTERVAL))
4.      AND (A1.quantity = C.quantity)
5.  Then Begin
6.      If  (A1.unit = C.unit)
7.      Then
8.              If  (A1.gran = C.gran)
9.              Then Begin
11.                             A3.quantity := A1.quantity;
12.                             A3.unit := A1.unit;
13.                             A3.gran = A1.gran;
14.                             A3.T  := A1.T;
15.                             A3.min := A1.min * C;
16.                             A3.max := A1.max * C;
17.                             build_scale(A3.min,A3.max,
                                                A3.gran,A3.V);
18.                             A3.EV := A1.EV * C;
19.                             A3.SD2 := A1.SD2 * C**2;
20.                             End
21.                     Else Begin
22.                             coerce_gran(A1,C, Aprime1, Aprime2);
23.                             Add_attribute(Aprime1, Aprime2, A3);
24.                             End
25.             Else Begin
26.                     coerce_unit(A1, C, Aprime1, Aprime2, error);
27.                     add_attribute(Aprime1, Aprime2, A3);
28.                     End
29.     Else ERROR( "Mismatched Scales")
30.     End; {add_attribute_plus_C}
```

Figure 3.8 — Attribute Semantics : (Times A_1 C) -> A3

Once again, the major difference is in statement 3, 18, and 19. Times_Attribute_C is also closely related to new_att_new_unit. The proof for statement 19 is as follows:

$$
\begin{aligned}
V(CX) &= E((CX)^{2}) - (E(CX))^{2} \\
&= C^{2}E(X^{2}) - C^{2}(E(X))^{2} \\
&= C^{2}[E(X^{2}) - (E(X))^{2}] \\
&= C^{2}V(X) \\
V(CX) &= C^{2}V(X)
\end{aligned}
$$

Figure 3.9 — Proof $V(CX) = C^{2}V(X)$

3.2.6 *Semantics of (Sub A_1 A_2) -> A_3*

In the discussion of dates in the previous chapter, it was pointed out that the difference of two interval scales results in a ratio scale. Consequently, when two interval scales are compared, they are compared in terms of the magnitude of the difference between them. In the case of current date and birth date, the difference results in a ratio scale age. In the Fahrenheit weather forecasting example, the difference between two temperatures can be used to state how much warmer or colder one day was than another. The semantics of subtraction are as follows:

```
(Sub A1 A2) -> A3
1.  Procedure subtract_attribute(A1, A2, A3)
2.  Begin
3.  IF(A1.quantity = A2.quantity) AND COMPARABLE (A1.T, A2.T)
4   AND ((A1.Scale = RATIO AND A2.Scale = RATIO)  OR
            (A1.Scale  = INTERVAL AND A2.Scale = INTERVAL))
5.  Then Begin
6.      If  (A1.unit = A2.unit)
7.      Then
8.              If  (A1.gran = A2.gran)
9.              Then Begin
10.                     A3.Scale := RATIO;
11.                     A3.quantity := A2.quantity;
12.                     A3.unit := A2.unit;
13.                     A3.gran = A2.gran;
14.                     A3.min := max(0, A1.min - A2.min);
15.                     A3.max := A1.max - A2.max;
16.                     build_scale(A3.min, A3.max,A3.gran, A3.V);
17.                     A3.T  := A1.T
18.                     A3.EV := A1.EV - A2.EV;
19.                     A3.SD2 := A1.SD2 + A2.SD2; {assume ind}
20.                     End
21.             Else Begin
22.                     coerce_gran(A1, A2, Aprime1, Aprime2);
23.                     Add_attribute(Aprime1, Aprime2, A3);
24.                     End
25.             Else Begin
26.                     coerce_unit(A1, A2, Aprime1, Aprime2, error);
27.                     add_attribute(Aprime1, Aprime2, A3);
```

```
28.                 End
29.     Else ERROR( "Mismatched Scales")
30.     End; {subtract_attribute}
```

Figure 3.10 — Attribute Semantics : (Sub A_1 A_2) -> A_3

The subtraction algorithm is similar to the addition algorithm but it has some major differences. The differences are explained in the following paragraphs.

Statement 4

Although all of the checks for quantity and transition that were present in attribute addition are also present in attribute difference, the difference operation is equally applicable to both ratio and interval scales.

Statement 10

This statement declares that regardless of which types of scales are involved the difference operation is performed upon, the result is still a ratio scale.

Statement 14

Because the result of the difference operation is a ratio scale, the minimum of the new scale is constrained to be no less than zero.

3.3 Naming Attributes $\mathcal{N}(C)$

What is it that makes one object unique or different from another object? One of the components of domain knowledge is the recognition of properties that can be used to differentiate objects. Although classes can always create unique system names for an object, some classes have natural names that have a semantic meaning within the context of an application domain. These natural names uniquely identify an object of a class and correspond to an endusers understanding of the class as well as the domain of interest.

Domain specific considerations cause individuals to be distinguished in a variety of different ways; phone numbers identify people for political polls, fingerprints identify people for the FBI, while Social Security numbers identify people for the IRS and many other institutions. Within the meta-model the attributes that name a class are called the naming attributes:

A subset, $\mathcal{N}(C)$ of the $\mathcal{A}(C)$ are the naming attributes. $A \in \mathcal{N}(C) \rightarrow \text{null} \notin \mathcal{V}(\mathcal{A})$

Definition 3.4 — Naming Attributes

Identity and naming are also common themes within computer science. For example, Common Lisp has several functions to test for equivalence. The question of whether or not two variables are equivalent depends upon how that information will be used. If one is concerned with identity then the low level function *eq* is used.

(eq x y) is true if and only if x and y refer to the same memory location

But if one is concerned only whether or not values are the same then the more general function (equal x y) can be used.[3]

3.3.1 Existence and Domain Specificity

The domain specificity of equivalence, and an observation on class existence, is illustrated with an example of the definition of a book. Certain classes exist regardless of the existence, or the lack of existence, of any particular application domain. Books, for example, existed well before libraries and library computer systems and they will continue to exist without respect to whether or not either libraries or computers exist in the future.

Books have a title, an author, a publisher, a copyright date, a cost, a weight (used for shipping purposes), and an ISBN. The ISBN is a ten digit number (consisting of a group identifier, publisher identifier, title identifier, and check digit) assigned prior to publication that was created by book publishers as a means of uniquely identifying books. The number is constructed in such a way that each ISBN represents one and only one publisher-title pair. A partial view of a book class as it exists without regard to the existence of a library is shown in Figure 3.11:

```
Book = Class

  ISBN : Nominal:  ID_Number;
  Title : Nominal Title_Type;
  Author : Nominal Person_Name_Type;
  Copyright_date : Gregorian_date (min 1800);
  Cost : Ratio_scale money in dollars (gran .01) (min 1) (max 75);
  Weight : Ratio_scale weight in pounds (gran 1) (min 1) (max 35);
                  •
                  •
                  •
```

Figure 3.11 — Book Class

Bookstores uniquely identify books by their (title, author) pair or their ISBN identifier. Libraries also identify books by their titles and author (or by a call number derived from the author and title), but a library's requirements are different than those of a person or a bookstore; when a library acquires a book it needs to be able to differentiate between what otherwise would appear to be identical copies of the same book. If the books are different editions, the library can append the date of the edition. A more general method is to add a copy number and form a new unique identifier that consists of a pair such as (ISBN, Copy_Number) or (Library_of_Congress_ID, Copy_Number).

3.3.3 Naming Attributes and Data Base Schemas

Naming attributes have long been studied within the data base community. The following sections explain the relevant portions of data base literature and its relevance to the meta-modeling methodology.

---

[3] Common Lisp actually has four equality functions: eq, eql, equal, and equalp.

Within the world of data base schema evolution, the term *primary key* is roughly equivalent to the naming attributes within the meta-modeling methodology. Data base designers [Date 86] use the term key to mean a set of attributes that uniquely identify a tuple within a relation. If more than one key exists, then the keys are called candidate keys. The term "primary key" refers to the particular candidate key that has been selected to identify the tuple.

Referential Attributes, Foreign Keys

The term referential attribute [Shlaer 88] or foreign key [Date 86] is used to refer to an attribute of a class which is a naming attribute of a different class.

A subset, $\mathcal{R}(C)$ of the $\mathcal{A}(C)$ are the referential attributes, or $\mathcal{R}(C)$.

$$(\mathcal{N}(C) \supseteq \mathcal{R}(C)) \vee (\mathcal{R}(C) \cap \mathcal{N}(C) = \phi)$$

Definition 3.5 — Referential Attributes

Within the meta-model, referential attributes are used to associate classes as described in Chapter 6.

Functional Dependencies

Functional dependencies are the primary system for capturing class semantics within relational database models. A complete discussion of functional dependencies can be found in [Date 81], [Ullman 80], [Maier 83] and [Date 84],and is beyond the scope of this paper. The following brief summary is included only for definitional purposes.

First normal form is attained when the value sets of attributes contain only atomic values. In the meta-model, all classes enforce first normal form. Second normal form is attained when every nonkey attribute is fully dependent on the primary key. Third normal form is an additional restriction that requires that every nonkey attribute be non-transitively dependent on the primary key. Boyce Codd Normal Form (BCNF) was developed to handle overlapping candidate keys. The case of overlapping candidate keys occurs when the intersection between the sets of attributes which formed the candidate keys is not null. BCNF is attained if and only if every determinant is a candidate key. A determinant is defined as an attribute or set of attributes upon which some other attribute is fully functionally dependent.

Although functional dependencies were designed to allow the relational model to capture additional semantics of the external world from which the model was derived, neither Boyce Codd Normal Form (BCNF) nor third normal form are necessarily desirable in the meta-modeling technique. Examples of when neither BCNF nor third normal form conceptualizations are desirable are common. Consider a person class which includes the attributes name, address, city, state, and zip code. In this example, address, city, state, and zip code are dependent upon name. Zip code is dependent upon address, city and state. City and state are dependent upon zip code. And yet, it is normal and customary to keep all of these attributes together in the same class regardless of the update anomalies which theoretically may occur. As another example, consider an order form in which subtotal and sales tax are stored along with the total. Once again, this is a normal way to

view an order form. But Sales Tax is Dependent on Subtotal (and State), Total is dependent on SubTotal and Sales_Tax. This, of course, violates BCNF.

3.4 Additional Restrictions on Attribute Values

Certain attribute restrictions are possible to specify at the class level, but impossible to specify at the attribute level without the context of a class. This is because these restrictions apply to the set of objects, all of which have the same attribute.

3.4.1 Requiring Unique Values

Unique values are an additional attribute restriction which can be applied at the class level. This restriction enforces the constraint that, for any new object created by the class, the value of the attribute will be unique within the set of objects created by that class. Naming attributes are always required to have unique values, and depending upon the domain semantics this restriction may be applied to other attributes.

3.4.2 Restricting Null Values

A null value is a special value assigned to an attribute to indicate that the value has not yet been determined. When an object is in the process of being created, many of its attributes might have the system assigned value of NULL. A decision to allow an object to retain null values for any of its attributes is in most cases domain specific. In general, for an attribute to be allowed to be NULL it must not be used to identify the object, must not interact with other objects and must not be used in a derived formula.

Null values are frequently problematic in information systems because of their inconsistent treatment and use. For example, in SQL, it is possible to have the average of an attribute not equal the sum divided by the count. This is because the count function returns the total number of records. A sum function returns the sum of a particular attribute and the average function returns the average of the non-null attribute values but excludes the nulls in the calculation.

The meta-model avoids this problem by keeping a consistent distinction between a summary statistic defined on the set of objects of a class and the summary statistic of a particular attribute within that class. This is further discussed in the section 3.6 which discusses the summary statistical functions $\mathcal{S}(C)$ and $\mathcal{S}(A)$.

3.5 Operations

Class operations exist to perform actions on classes. Two special operations are the procedures for object instantiation and deletion.

Classes include:

An instantiation procedure I that uniquely associates any object of class C with some tuple of $\mathcal{V}(N_0)$ X . . . X $\mathcal{V}(N_n)$ $(N_i \in \mathcal{N}(C))$.

A set of deletion or removal procedures , $\mathcal{R}(C)$, that remove the objects created by the instantiation procedure I.

A set of Operations, $O(C)$, that access or update sets of attributes

Definition 3.6 — Operations

One of the purposes of the class structure is to be able to instantiate and delete objects whose type has been defined by the class. *Instantiation* and *deletion* are fundamental operations capture semantics in terms of the meta-modeling technique. Figure 3.12 gives some of the interpretations for these terms as they appear in business application domains.

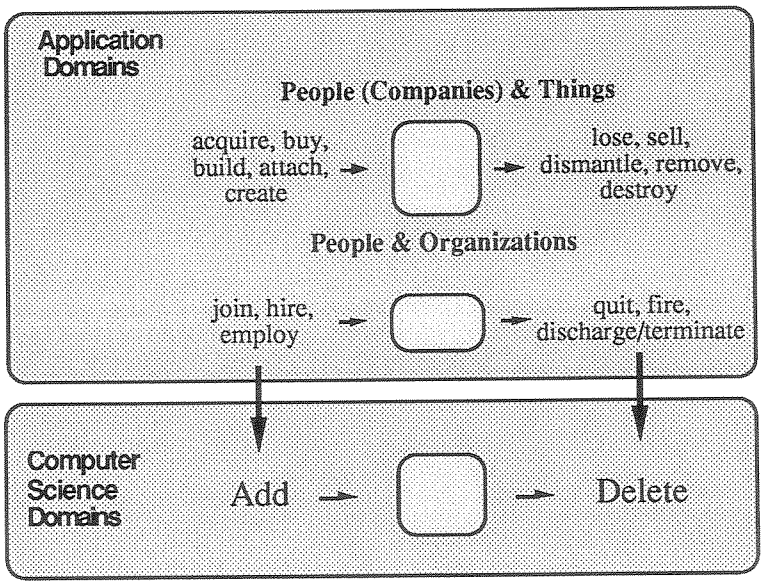


Figure 3.12 — Domain-Specific Interpretations of Add & Delete

3.5.1 Instantiating Objects

Adding an object to an application domain is accomplished by the class creation procedure. As was previously discussed, the selection of a name—the primary key—for an object is determined by the domain expert's interpretation of an application domain. The class instantiation procedure must be able to create a name in such a manner that an object can be located, accessed, and modified during its life in the system. This requires that the object have some type of unique identification that differentiates it from other objects for the purposes of system identification. Within a particular domain model, the instantiation procedure I stores the domain knowledge about how to instantiate objects.

The peculiarities of application domain knowledge force different classes to be defined for the same entity when these entities are used in different domains. As was described, an

ISBN is sufficient to uniquely identify a book for a book publisher of a bookstore, but not for a library. Libraries require additional attributes such as copy number because they need to keep track of individual books.

In actual practice, the enduser has a great deal of control over the creation of an object within a class. Imagine an enduser at a terminal filling in a data entry screen. For each object entry screen (more details given in [Iscoe 86]), endusers are able to completely finish filling out an object instantiation, cancel an object definition, and request help functions. The following system events are predefined for all object instantiations.

Objectinit *When the object first appears,*

Fileinit *When the enduser first start as session,*

Objectdone *When the enduser finished filling out the object entry screen,*

Objectcancel *If the enduser abandons filling out the object entry screen,*

3.5.2 Deleting Objects R(C)

Although deleting an object is the logical inverse of instantiating that object, additional complications are introduced for object deletion. Consider the case of a member quitting a library. If the member has books that are checked out, then a problem is created; How should the system handle the books that are still checked out by the member? When one object is linked to another object through a referential attribute, the fate of an object has reverberations to all other linked objects within a system. Within the data base community there are three standard methods for handling the deletion of a object that contains references to other objects.

Cascading deletions -- Deleting an object causes the objects that depend on it to be deleted. If an object *b* is referenced by an object *a*, and *a* is deleted, then object *b* (and all objects dependant upon it), should be deleted. This process continues (cascades) until no more objects are effected.

Nullifying effected fields -- Objects that are effected by the deletion have their relevant attribute values set to null. The deletion of *a* should be allowed to happen but that the pointer to object *b* should be set to null.

Restricting deletions -- Object is not allowed to be deleted. A restricted deletion means that if object *b* depends upon object *a* and object *b* exists, then object *a* should not be allowed to be deleted.

These three strategies are reasonable when one considers the effect of deletion on a static set of data. However, in an application program that includes objects that have an independent existence in the real world, none of these strategies may be appropriate. For example, in the library problem, above, neither cascading the deletions (and eliminating the books) nor nullifying the effected fields (and losing track of who checked out the book) are acceptable operations. And yet, the member has left the library. From a strictly database point of view, we are tempted to restrict the deletion until all is resolved (i.e. the books are declared lost or returned), but this is begging the question of what to do in an application program.

Deletion procedures maintain domain knowledge about events such as the procedure to follow in the event of a lost or stolen library book.

3.5.3 Other Operations

In addition to defining instantiation and deletion operations, the class definition serves as the repository for the definition of operations that access or update attributes of a class. These operations, sometimes called methods in object-oriented terminology, can be applied to all objects instantiated by a class. In Chapter 2, single attribute operations were discussed. Because classes have now been defined, we are in a position to discuss the generalized access and update operations on multiple attributes.

3.5.4 Access & Update Operations

Access operations are those operations which retrieve information, which may be based on a series of selection or restriction conditions. For example, in the library problem one of the specifications is that the following request must be satisfiable.

> *T3. Get the list of books by a particular author or in a particular subject area;*

- get_info(*class_name [restricticti on_list] [selected_list]*)
- get_info(book_class (author x) [book_list])

Figure 3.13 — Class Access Operations

From the standpoint of the domain model, the important consideration is that this is a query that identifies individual objects by some combination of their attributes. As such, it represents the most trivial – but possibly most common – type of information retrieval within an application program.

It should be noted, that there are many different ways that these queries could be expressed. This request is a standard data base style query that could be expressed in SQL, QBE, prolog, or any of a number of other styles.

Update operations are those operations that are used to update an attribute or a set of attributes. The update operation move is illustrated in the following example.

3.5.5 Example — latitude and longitude revisited

The F16 example that was introduced in Chapter 1 illustrated a specification problem caused, most probably, by a mapping mistake in which the latitude definition was replaced by a signed number. As can be seen from figure 3.14, the algorithm for maintaining correct direction given any particular move allows latitude and longitude to work as specified. Latitude is defined as the angular distance, measured in degrees north or south from the equator. Longitude is defined as the angular distance east or west on the earth's surface measured by the angle (expressed in degrees up to 180° in either direction) which the meridian passing through a particular place makes with a standard or prime meridian, usually the one passing through Greenwich, England.

Clearly the magnitude of latitude is a ratio scale. There are no negative latitudes, and the zero of the latitude has a real meaning (i.e., it is the point at which north changes to south). Two attributes are needed to represent latitude; the first attribute is a ratio scale that

represents magnitude and the second attribute is the nominal scale that represents north or south. Longitude is represented in a similar manner.

In chapter 5 after composition has been introduced, an airplane location class will be defined that combines altitude, latitude and longitude.

```
Navigational Location = Class

    Latitude.direction: Nominal (North .5) (South .5)
    Latitude.magnitude: Ratio Angle in Degrees (min 0) (max 180)
    Longitude.Direction: Nominal (East .5) (West .5)
    Longitude.Magnitude: Ratio Angle in Degrees (min 0) (max 180)


Operation move.[latitude | longitude] (compass, distance)
    If compass = #.direction
    then
            begin add(#.magnitude, distance, error, error_amt)
                    if error then begin
                            change_state(#.direction);
                            set_value(#.magnitude, error_amt)
                            end
    end
   else
     begin sub(#.magnitude, distance, error, error_amt)
            if error then begin
                    change_state( #.direction);
                    set_value(#.magnitude, error_amt)
                    end
    end

end; operation-move
```

Figure 3.14 — Latitude and Longitude Class

3.6 Summary Statistical Functions

> Classes include a statistical function $\mathcal{S}(C)$ that computes summary statistics for the set of objects instantiated for C, and a set of functions $\mathcal{SA}(C, A_i)$ that compute summary statistics for each attribute of the set of objects.

Definition 3.7 — Class Statistical Functions

Summary statistics characterize information about the objects created by the class. Two types of summary statistical functions are defined for classes. The functions $\mathcal{S}(C)$ compute summary statistics for the overall set of objects created by the class, and the functions $\mathcal{SA}(C, A_i)$ compute statistics for each individual attribute of that class.

3.6.1 *Class Summary Statistics $\mathcal{S}(C)$*

The summary statistics that the class maintains for the object set reflect information about the function count (t) of the number of objects in the system at time t. The summary statistics concerning the count of an object set are the average of that count, the standard deviation of that count as well as the mode, minimum, and maximum of that count.

These are all easily computed by maintaining the count, sum, and sum of square for each class over the life of a system. Additional statistics such as median about the objects in the object set requires some notion of system history.

3.6.2 *Attribute Summary Statistics $\mathcal{SA}(C, A_i)$*

For any particular attributes of a class, the kind of attribute statistics available is dependent upon the scale type of the attribute and were summarized in Figure 2.6 of Chapter 2. The $\mathcal{SA}(C, A_i)$ are statistics of the sample of attribute values represented by the existing objects created by class at any particular point in time. These are statistics derived from the sample of attribute values represented by the existing objects of the system.

3.6.3 *Attribute Defaults Revisited*

In the last chapter, default values were discussed in terms of attribute population parameters. Because we have now introduced classes, we are able to introduce a more sophisticated system for the calculation of missing information. Using population parameters, it is possible for the domain model to be able to *deduce* a missing value for an attribute of a particular object.

This technique assumes that classes have been partitioned in a careful enough manner that they truly characterize the application domain[4] (a further discussion of this point). Sociologists, marketing companies, and other survey analysts have long used this type of system to supply missing information. Essentially it works by using values known about an object to infer other values of that object. For example, a person object might be partially defined as follows:

```
Person = Class
    Name: Person_name;
     SSN: nominal_scale;
            •
            •
            •
    residence: nominal_scale (rent, own);
    student: nominal_scale (yes, no);
    age: ratio_scale. (min 18) (max 99);
```

Figure 3.15 — Partial Definition of Person Object

In this case if the person is: a student, under 25 years of age, and the type of residence is left blank, the default mechanism supplied by the modeling methodology could, for this domain model, automatically assign the the value *rent* to the residence attribute field.

---

[4] This issue is further mentioned in Chapter 4, and discussed in detail in [Kish 65] and [Rosenberg 68].

However, if the object is in a different partition such as if the person is a student and is over 25 years of age, the default mechanism would not be able to assign a value based on the preceding information, and would have to examine other attributes such as income in order to be able to fill in the missing information.

It should be noted that the continual use of information in this manner creates a phenomena known as regression towards the mean. The term *regression towards the mean* refers to the phenomenon that is created through the continual use of expected values to supply missing values. This process creates expected values which, as the default mechanism iterates, progressively draw closer and closer to the mean.

3.7 Class Axioms χ(A)

Classes include a set of axioms or integrity constraints χ(C):

- interclass attribute axioms
  - intraobject attribute axioms
  - interobject attribute axioms
- interclass attribute axioms

Definition 3.8 — Class Axioms

Class axioms are a mechanism for supplying constraints to attributes within classes. The meta-model organizes the axioms by grouping them into categories. One useful categorization is to split the axioms into disjoint groups:

- Those that work within a particular class (the intra class axioms); and
- Those that work between classes (the inter class axioms).

The intraclass axioms can be further broken down into the intraobject attribute axioms and the interobject attribute axioms.

3.7.1 Intraobject attribute axioms

Intraobject axioms define relationships between the attributes of the runtime objects of a particular class. For example,

Tax = Total * TaxRate

specifies an invariant relationship between attributes within a particular object from a class. It should be noted that this type of integrity constraint is related to but not equivalent to the situation in which tax is defined as a derived attribute that is computed from the primitive attribute, total and tax rate.

There are many occasions in which the situation occurs in which an integrity constraint axiom is used instead of a derived attribute. In data entry applications, the classic example is that of a check digit. Check digits are computed by an algorithm and compared against information entered by the enduser. In the tax example, an application designer might choose to use an axiom to check a tax that had already been computed by a person or by another system.

At the detailed level of an application design, many situations of this nature arise. There are other circumstances when the value of one attribute dynamically determines

whether or not another set of attributes is applicable. For example, if a company is tax exempt then its tax identification number is required.

```
if Tax_Exempt
        then Must_Enter (Tax_Identification_Number)
        else close_field (Tax_Identification_Number)
```

If the method of shipment is UPS, then there are three different choices for type of shipping and so on.

As was discussed in the introduction, the purpose of the meta-modeling methodology is to specify information at a high conceptual level. In previous work [Iscoe 86] a number of operations were described for the CRTForm type manager and associated LKP system. These functions were implemented through a type manager, and served to add additional dynamic constraints to an attribute definition. While some of the constraints such as entry sequencing were specified interactively, other types of axioms were implemented as type manager operations and are the types of constraints that would be used by an application designer and not by a domain expert. The type manager operations used in the previous research that are relevant to this research are shown in Figure 3.16.

`Erasefield` (Field); removes field from the screen.
`Gotofield` (Field); forces the cursor to a specific field.
`Loadfield` (Inputdata); loads a string into the Readfield buffer.
`Lockfield` (Field); protects a field from entry by the user.
`Lockset` (Set_ID); protects a set of fields from entry.
`Resetfield`; clears and resets a field.
`Sendboolean` (Field, Flag); sends a Yes or No to a field.
`Senddate` (DD,MM, YY); sends a date to a field.
`Senderror` (Bellpattern, Attribute, Message); writes a message and locks the keyboard.
`Sendmsg` (Bellpattern, Attribute, Message); writes to the bottom line.
`Sendstring` (Field, Message); sends information to a field.
`Showfield` (Field); activates fields which are in memory but not displayed.
`Showset` (Set_ID); activates fields which are in memory but not displayed.
`Showlines`; displays line drawings of form on screen.
`Unlockfield` (Field); changes a field from protected to unprotected.
`Unlockset` (Set_ID); changes a set of fields from protected to unprotected.
`Videoinput` (Field, Attribute); changes video attributes of field input.
`Videotitle` (Field, Attribute); changes video attributes of a title.

Figure 3.16 — Relevant Type Manager Functions

3.7.2 Interobject attribute axioms

Interobject attribute axioms constrain the value of an attribute by comparing its value to the values of the set of objects of that class. The values of a set of objects were previously defined in the section on summary statistics. These values are derived from the functions

S(C). For example, this type of axiom could restrict an employee's salary to be no greater than forty thousand dollars more than the average salary.

Emp.salary< mean(emp.salary) + $40,000

3.7.3 Interclass axioms

Interclass attribute axioms are axioms that restrict the value of an attribute of a class by referencing the value of an attribute or attributes or the value of a system function from another class. For example, one such axiom might be:

maximum_number_of_books_checked_out < count (books) / count (number of people).

There are at least two interpretations of the previous axiom. One interpretation is that this constraint must hold true only at the time of checkout of the book. Another interpretation is that the constraint must be maintained throughout the life of the system. Obviously, the second constraint requires a great deal more overhead in terms of implementation.

3.8 Transformational Implementation Issues

The runtime characteristics of a class can help to determine the type of storage, the type of indices, the methods of retrieval, and other aspects of an implementation. For each class the following questions can be answered by an application designer:

- Current number of objects in a system?
- Current number of objects added per period?
- Objects deleted per period?
- Percent growth rate per period equals (#added per period - #deleted per period)/ (#of objects in system).

This information, when combined with the specification information for a class, allows a number of optimizations to be performed by the transformational implementation of the final application program. The type of scenario that an application designer might describe for a library is as follows:

> *The annual renewal membership rate for people at the library is 85% and we anticipate a 25% per-year growth rate. The library receives 4,000 books a year and loses about 750 through theft and damage. We currently own about 60,000 books.*

Current_size[5](person_class) * Estimated_max_number(person_class)
Current_size(book_class) * Estimated_max_number(book_class)
summation of (size(obj) * max_expected(obj)) < disk capacity.

---

[5] Where current_size is used to indicate bytes, block size, or whatever measure is most appropriate for the implementation.

Figure 3.17 shows the screen that captures class population parameters. As was shown in Iscoe [Iscoe 88a], this information can be used to facilitate selection of the appropriate database algorithms for an application program.

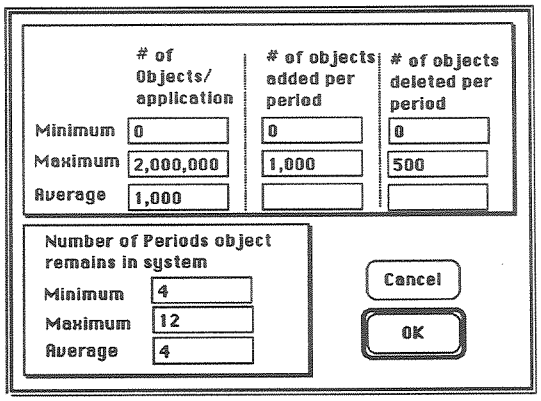


Figure 3.17 — Class Population Parameters

3.9 Organizing Classes

Classes have been defined as constructs that encapsulate attributes, provide for the definition of derived attributes, allow additional operations to be defined, and at run time instantiate and delete objects as well as maintaining summary statistics and enforcing integrity constraints. Several examples were used in this chapter to demonstrate both the definitional aspects of domain models as well as the methodology for instantiating domain models.

The definition of class combined with the definition of attribute in the previous chapter completes the building block definitions for the meta-model. Attributes can be viewed as abstract data types that capture domain semantics which are normally lost when standard computer types are used to represent domain values. Classes are higher level abstractions that encapsulate a set of attributes and provide a structure for capturing additional domain information.

The next three chapters describe how information stored within the attribute/class framework is used by domain experts to organize and structure domain models. The main class structuring concepts are hierarchical decomposition, class composition, and class association. As was mentioned in Chapter 1, many object-oriented and knowledge representation schemes use these concepts. However, our methodology is more definitional, more strongly typed, and more structured than most other approaches. This is possible because we have traded structure for exploration. The emphasis in this methodology is on providing a meta-model that domain experts can use to record their knowledge instead of an exploratory system in which novices make discoveries about an application domain. This tradeoff allows us to avoid many of the ambiguities and confusion such surrounding many models.

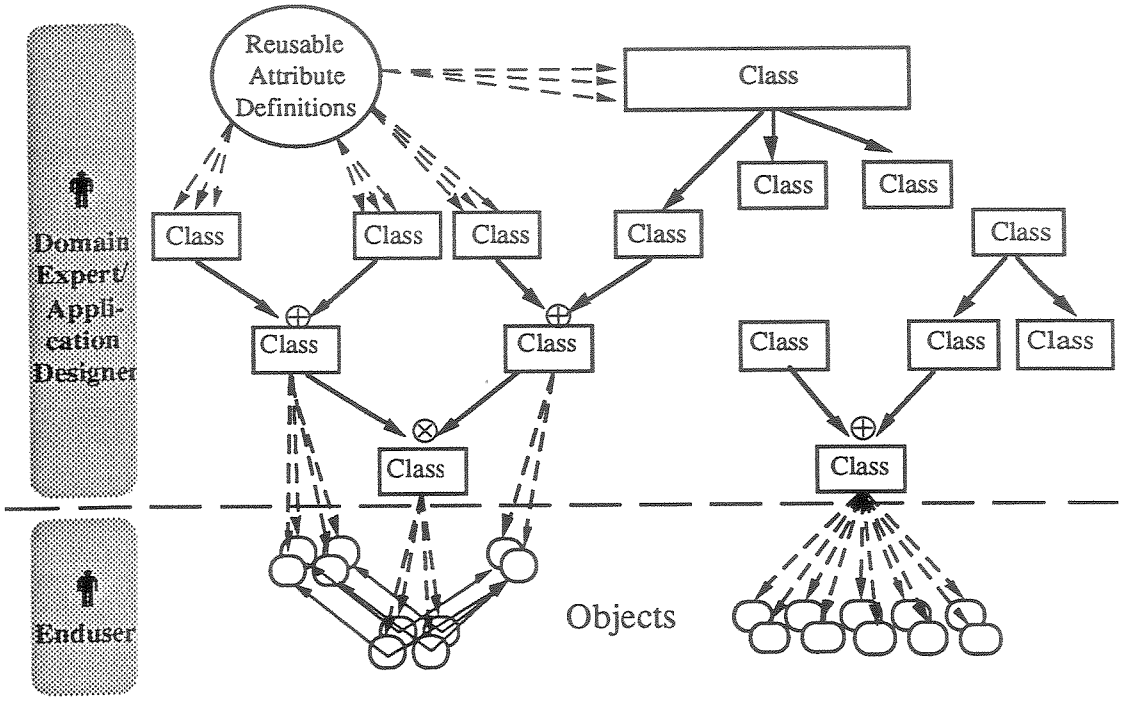


Figure 3.18 — Overview of the Model

Figure 3.18 shows attributes, classes, hierarchical decomposition, composition (indicated by symbol ⊕), and association (indicated by the symbol ⊗) Hierarchical decomposition is the process of developing a class hierarchy by using *attribute restriction* to *specialize* a class into *subclasses*. Sub (and super) class definitions, along with our interpretations of the terms *inheritance, generalization, specialization, aggregation, and instantiation* are explained in Chapter 4.

Class composition is the process of creating a new class from two or more other classes by taking the union of their sets of attributes and using domain knowledge to resolve conflicts, eliminate unnecessary attributes, and add new attributes as needed. By creating classes that represent common entities or concepts within an application domain, the domain-expert is able to build a library of primitive classes that can be used, and reused, in a variety of combinations to create larger-grained classes. These ideas are further explained in Chapter 5.

Chapter 6 defines class association; the process of creating a new class that establishes a relationship between at least two other classes by taking the union of their naming attributes and using domain knowledge to add new attributes as needed.

Chapter 4 - Hierarchical Decomposition

4.1 Introduction

Hierarchies are a natural way to view and organize information and, at some level of abstraction, are a part of most object-oriented and knowledge representation languages. Unfortunately, the simplicity of these concepts can sometimes obscure the semantics that a model is attempting to capture. We avoid unnecessary ambiguity [1] in the modeling methodology by forcing domain experts to define class decomposition in terms of an existing well defined attribute as described in definition 4.1.

C' is a subclass of C if:

1. There is an attribute, A, with $\mathcal{V}(A) \supset \mathcal{V}'(A)$
2. $P(C) \supseteq P(C')$ That is, the attributes of C' consist of at most the primitive attributes of C and additional attributes that can be derived from those primitives.

C is a superclass of C', if C' is a subclass of C

Definition 4.1 — Subclass and Superclass

In general, domain experts create subclasses by selecting an attribute of the superclass and then using its partitioned value set to defines subclasses. Subclasses contain only those attributes defined in the superclass, or that can be derived from the attributes of a superclass. *Specialization* is the process of partitioning a class on a particular attribute that is chosen to be appropriate by the domain expert. Figure 4.1 shows a company class that has been subdivided into three *subclasses* based on the value of the attribute, days_overdue. The three *subclasses* classes are *specializations* of the parent class derived by partitioning the parent class on the value set of the attribute *days_overdue*. The *superclass* company class is a *generalization* of its subclasses.

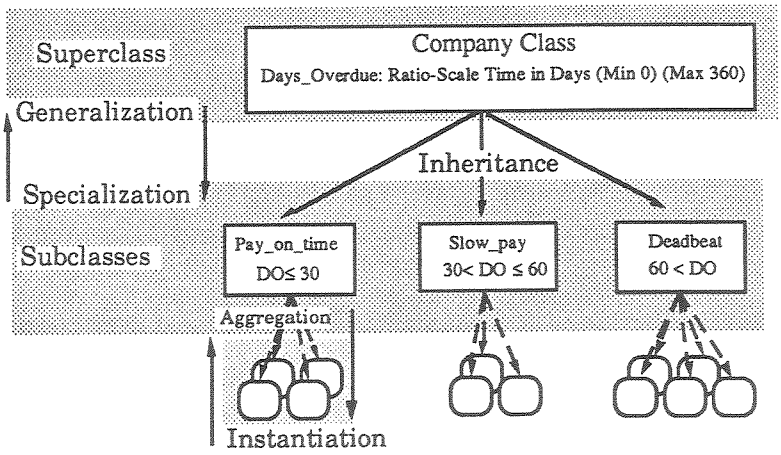


Figure 4.1 — Partitioning a Company Class

[1] Such as the multiple interpretations of "is-a" discussed by [Brachman 83].

4.1.1 Attribute Restriction $A|_{F, V, B}$

Throughout Chapter Two, we developed a system for defining attributes by systematically characterizing their value sets. In this chapter, we continue the process of characterization. By creating a systematic methodology for the partitioning of attributes, the modeling methodology provides a way for the domain expert to be able to organize, subdivide, and build hierarchies of classes.

Starting with the base definition of an attribute A, it is useful to define a restriction B as follows:

An attribute B is a restriction of an attribute A (or A is an extension of B) if :

$\mathcal{V}(A) \supset \mathcal{V}(B)$

$\mathcal{R}(A) \supset \mathcal{R}(B)$

Definition 4.2 — Attribute Restriction

Restrictions allow hierarchies of attributes and scales to be constructed. For example, the attribute car_color based on the nominal scale colors *(red, yellow, green, blue)* can be further divided into two restricted attributes:

$\text{Car\_color}|_{warm}$ and $\text{car\_color}|_{cool}$

where $\mathcal{V}(\text{car\_color}|_{warm}) = (\text{red, yellow})$

$\mathcal{V}(\text{car\_color}|_{cool}) = (\text{green, blue})$

Domain experts and application designers can design restrictions based on scale characteristics that include population parameters as described in the following section.

An attribute restriction was defined as a new attribute whose value set and set of applicable relations were subsets of the original attribute. In the car_color example, car_color was a four-category nominal scaled attribute that was mapped into a two-category nominal scaled attribute. In Figure 4.1, days_overdue was a ratio scaled attribute that was mapped into a three-category nominal scaled attribute based on cut-off values determined by the domain expert.

The modeling methodology decomposes classes by selecting an attribute and then partitioning its value set to create new categories. The process of partitioning a class generally results in creating a newly derived attribute that represents the mapping from:

- an interval or ratio scaled attribute to an ordinal or nominally scaled attribute.
- an ordinally scaled attribute to an ordinally scaled attribute with fewer categories than the original or to a nominally scaled attribute.
- a nominally scaled attribute to another nominal attribute with fewer categories than the original attribute.
- an interval or ratio scaled attribute to an ordinal or nominally scaled attribute.

For example, days_overdue is a ratio scaled attribute that was reduced to a three-category ordinally scaled attribute In the car_color example, car_color was a four-category

nominal scaled attribute that was restricted to become a two-category nominal scaled attribute that indicated color warmth.

4.1.2 Inheritance

Inheritance is the process by which a subclass receives the attributes of a super class as described in definition 4.1. This definition of inheritance differs from others in that it exists for definitional, as well as, implementation purposes. This approach is much more structured than other approaches commonly used with object-oriented programming. It requires that when a domain expert adds additional values to an attribute value set that those values will be propagated to the attribute dictionary and available to all classes as opposed to being only defined locally at a low level in a class hierarchy.

One of the goals of inheritance is to permit a disciplined form of breaks in the veil of encapsulation for classes. This break is necessary because as the number of classes increases (and from a designer-programmer standpoint it becomes more important to see inside a particular class), programmers often cheat. That is, they directly reference instance variables or use a variety of other techniques to pierce the encapsulation veil.

Snyder [Snyder 88] discusses the dynamic tension that exists between encapsulation and inheritance. From a software engineering standpoint, object-oriented approaches are good because they hide detail by encapsulating state and operations, but the benefits of being able to "see inside" a class or an object cause many designers to decide that strict encapsulation is overly restrictive. In the modeling methodology a balance is achieved between encapsulation and inheritance by reusable attribute definitions that provide common building blocks for class construction.

4.1.3 Aggregation

Aggregation is a frequently used term that refers to the construction of a class based on observations about object clustering within a particular application domain. Figure 4.1 shows the term *aggregation* as pointing up from a group of objects to the class that might have been created through aggregation. Although aggregation is a term used in a variety of other approaches, we do not use the term in the modeling methodology. This is because aggregation is a process that has presumably been performed by the domain expert to produce the conceptual model of the domain. While aggregation is an implicit part of the domain structure, the non-exploratory nature of our methodology precludes aggregation from being a class construction primitive.

4.1.4 Instantiation

The term instantiation is used in this dissertation to describe two different processes; one process occurs at the meta-level and the other process occurs at the domain modeling level.

At the meta-level, as was discussed in Chapter 1, the term instantiation describes the process in which a domain expert instantiates the meta-model with domain information to create a specific domain model. At the domain modeling level, classes instantiate objects which represent instances of the type defined by the class.

4.2 Using Population Parameters

Population parameters facilitate the formation of new attributes. For example, some graduate admissions committees use interval scaled GRE scores to separate applicants into acceptance categories. Population parameters allow designers to create new attributes based on restrictions to the original attribute as shown in Figure 4.2.

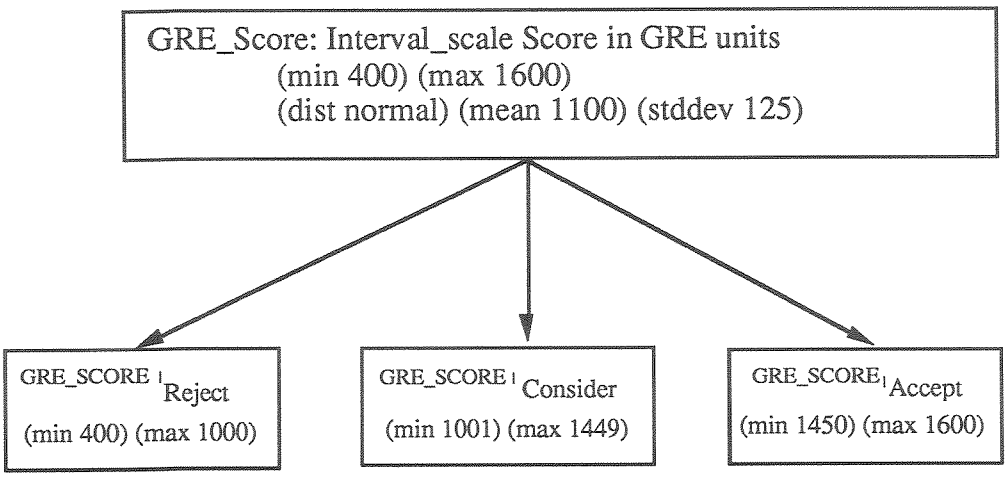


Figure 4.2 — Using GRE Score to partition a class

Figure 4.2 shows the GRE score as an attribute which could be attached to a student. Understanding the distribution of values within the value set of GRE scores allows application designers to create partitions in any one of a variety of ways. For example, assume that an application designer wanted to create an initial partition based on the requirement *"accept all students who score in the top x% on the GRE and reject those who score in the bottom y%"* Given this type of requirement, the domain model contains the appropriate information to use and algorithm to produce the correct raw score numbers to achieve such a partition.

Another way that these requirements are sometimes stated is to build a partition based on an absolute raw score. For example, a requirement like, "*accept all students who score above 1450 on the GRE*" can be easily incorporated. Furthermore, this type of specification can be used interactively so that the designer can juggle between raw scores and percentiles until the partitions appropriate for the application domain are produced.

4.3 Hierarchical Decomposition & "Birds Fly"

While our approach to domain modeling has more restricted operational goals than the generalized research of knowledge representation, it is interesting to use this methodology on the classic problem of representing *birds fly*. This example usually begins with the assertion that all birds fly, $\forall x$ (bird(x) $\rightarrow$ fly(x)).

Using the modeling methodology, the first task is to create an attribute (in this case nominal) whose value set represents all possible states of the attribute. Because can_fly has been specified as one state of flight ability, the other option to consider is the absence of flight, can_not_fly.

flight_ability: nominal_scale (can_fly, can_not_fly)

In a simple system, this attribute is then assembled with a bird_type attribute to form the bird class shown in figure 4.3a where the value set of flight_ability would be constrained to the single value can_fly.

```
Bird = Class
    Bird_type: (robin, sparrow,penguin, ostrich, . . .)
    flight_ability: nominal (can_fly, can_not_fly )
```

Figure 4.3 a — Bird Class

Once the initial premise ($\forall x$ (bird(x) $\rightarrow$ fly(x))) is assumed to be true, a series of counter examples are then presented. These questions include the following: What about penguins? Ostriches? Birds with broken wings? Dead birds? Birds with their feet set in concrete? The problem then is to identify the no longer correct assertion $\forall x$ (bird(x) $\rightarrow$ fly(x)) and to retract it along with the facts proved from it.

One way to handle these issues is shown in figure 4.3b. By adding population parameters, the model addresses the fact that bird(x) neither implies flight or non-flight. The class no longer contains incorrect information, and the problem posed above is avoided.

```
Bird = Class
    Bird_type: (robin, sparrow,penguin, ostrich, . . .)
    flight_ability: nominal ((can_fly 75)  (can_not_fly 25))
```

Figure 4.3 b — Bird Class with Population Parameters

Our methodology allows us to sidestep the inference question that the logicians are actually posing, and encourages the domain expert to approach the problem in a different manner. Notice that the questions posed above fall into two different categories. One set of questions addresses issues concerning a taxonomy, while the other set refers to operational considerations of bird flight.

Chapter 2 described $\mathcal{PC}(A)$, the probability that an attribute will change value within the life of a system. In the event that an attribute does not change value within the life of an application, $\mathcal{PC}(A) = 0$. Examination of this characteristic is particularly appropriate when the domain expert is creating taxonomic partitions, because attributes that are allowed to change should not be used for taxonomic classification purposes. Obviously, the term flight_ability has been overloaded to mean both the inherent ability to fly as well as the operational ability to fly, and was therefore an inappropriate selection to achieve a taxonomic classification.

Figure 4.4 shows a second attempt at creating a workable class structure. If one cares about the operational ability of birds to fly, then that ability should be represented by a (nominal scaled) attribute, flight status. Given that a bird flies it might have two statuses: A-O.K. means that the bird can fly, while grounded might mean that the bird has a broken wing, is sick, doesn't want to fly that day, is dead and so on. On the other hand, even birds that can't fly might have occasion to hitch a ride on a airplane, on the back of a larger bird, or perhaps be tossed through the air by a young child.

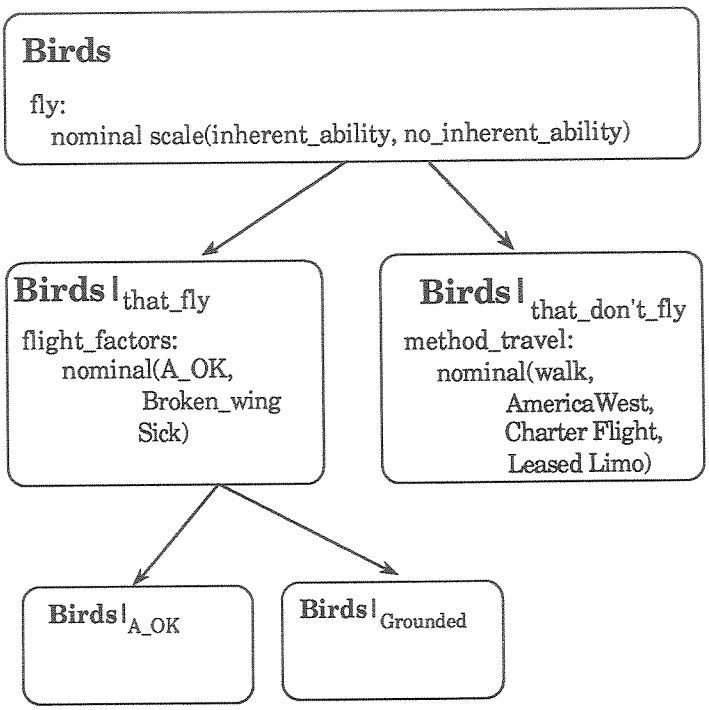


Figure 4.4 — Birds Fly

While this method of representation doesn't solve the birds fly inference problem as it is normally posed, it illustrates operational specifications as used within the modeling methodology. In this case, the original attribute used to partition the class of animals was not appropriate because it failed to provide a taxonomic partition. By recognizing the two components of fly, an operational solution was achieved.

4.3.1 Definitional Components

[Brachman 85b] has pointed out that many of the cancellation and default schemes used in knowledge representation can lead to situations in which a model can be interpreted in obscure and non-constructive ways. He illustrates this by observing that the concepts of 3-legged-elephants and 4-legged-elephants are sets of properties that are usually not defined in a way that establishes what the normal or usual situation should be. Figure 4.5 shows a definitional solution to that problem using the modeling methodology.

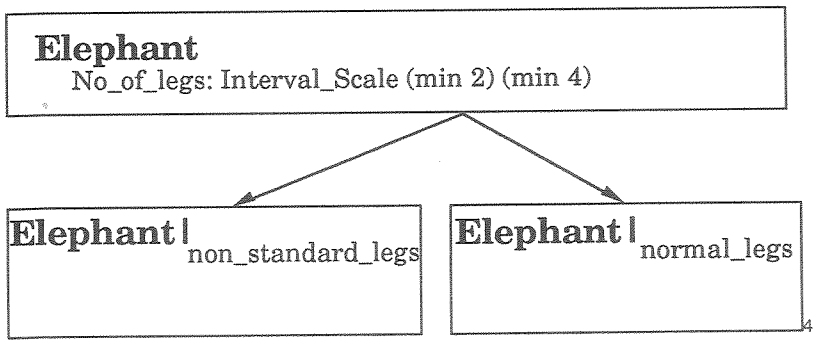


Figure 4.5 — Elephant Legs

4.4 Hierarchy Evolution

This section is a discussion of how a domain expert evolves a domain model. Assume that a domain expert has defined a book class as one which includes attributes such as author, title, subject, ISBN, and so on. Let us now assume that when examining the book class, the domain expert wishes to include a designation for a paperback book. The first step is to determine whether the paperback designation is a fundamentally new attribute or simply an extension of an old one.

An example of attribute extension would be adding an additional color to the book color attribute, an additional subject category to book subject attribute, and so on. In order to add the paperback designation, the domain expert must decide if this is a new classification that wasn't originally considered part of the domain, or if it can be incorporated into a previously existing classification. One way to approach the problem is to consider the question: "What does it mean to not be a paperback? Or, "Does this attribute partition the physical characteristics of the class?" This question relates to the original reason that the expert felt that paperback was an important attribute. Perhaps they are cheaper, or easier to get, get stolen more often, don't last very long, or some combination of these reasons.

This discussion points out that the decision to include or exclude attributes is domain-specific. If the reason that a paperback classification is important is that it will be used for shipping purposes, then the paperback designation might more appropriately be a specialization of book weight. Similarly, if the purpose of the paperback designation is to indicate a low cost, then a specialization could be performed based on book cost for books under $10.00. However, if the paperback classification is in and of itself of interest, then a new attribute might need to be created. This attribute might also include other values which were not originally included such as oversized books, books with photographic plates, and other unusual books.

Chapter 5 – Composition

5.1 Introduction

Composition is the basic constructor of the domain modeling methodology, and is the process of constructively building new classes from previously existing ones by using domain knowledge to:

1. take the disjoint union of the sets of attributes,
2. resolve conflicts,
3. eliminate unnecessary attributes,
4. add new attributes as needed.

The prefix symbol $\oplus$ is used to indicate the composition process as in the expression: $C_M := (\oplus\ C_1\ C_2 \ldots C_n)$. The following definition gives the basic algorithm for the composition process.

C_M is a class that is the composition of the classes $C_1\ C_2 \ldots C_n$. The algorithm (in which the subscript ✍ refers to a function completed by the domain expert) is as follows:
$C_M := (\oplus\ C_1\ C_2 \ldots C_n)$

1. $\mathcal{A}(C_M) := \mathcal{A}(C_1) \cup \mathcal{A}(C_2) \cup \ldots \mathcal{A}(C_n))$, where $\cup$ is the disjoint union

2. $\forall x\ \exists y\ ((\mathcal{N}(\mathcal{A}(C_x)) \cap \mathcal{N}(\mathcal{A}(C_y)) \neq \varnothing) \rightarrow \text{conflict\_resolution}_{✍}(\mathcal{A}(C_x), \mathcal{A}(C_y))$

3. $\mathcal{A}(C_M) := \mathcal{A}(C_M)$ - irrelevant _attributes$_{✍}(C_M)$

4. $\mathcal{A}(C_M) := \mathcal{A}(C_M) \cup$ selected_new_attributes$_{✍}$(attribute_library)

Definition 5.1— Composition Process Algorithm

Throughout the remainder of this section, composition is illustrated with a series of examples that show how various types of information are combined, how conflicts are resolved, and how composition differs from multiple inheritance.

5.2 Composition without conflicts

Before beginning the detailed analysis of composition, it is useful to examine two figures introduced in chapter 1. Figures 5.1a shows classes in the context of a simplified library system, and figure 5.1b shows a simplified accounts receivables system. Explicit in these figures is the idea that certain classes exist regardless of the existence (or the lack of existence) of a particular application domain. Books, people, products, and companies, all exist in a variety of different contexts and are labeled as general classes in the figures. While domain boundaries vary, certain classes such as lending_information, member_information, pricing_information, and so on can be viewed as *domain-specific* classes because they are always bound to specific domains. The gray areas labeled association classes will be discussed in Chapter 6.

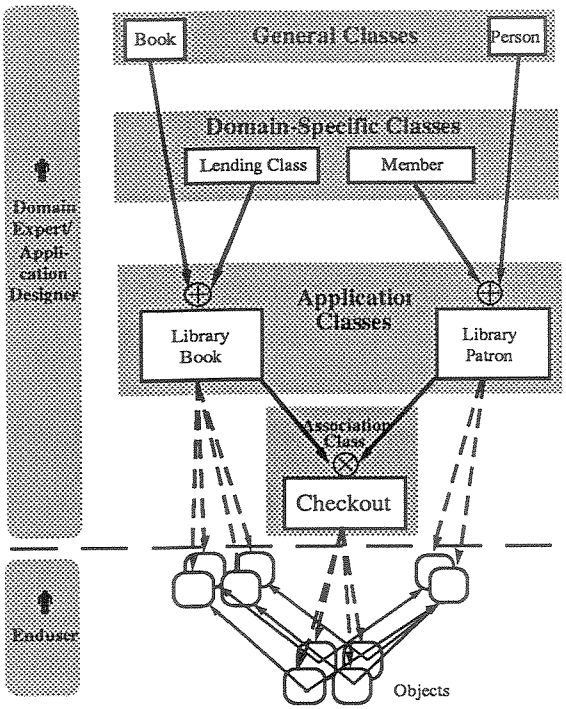


Figure 5.1a — Library System

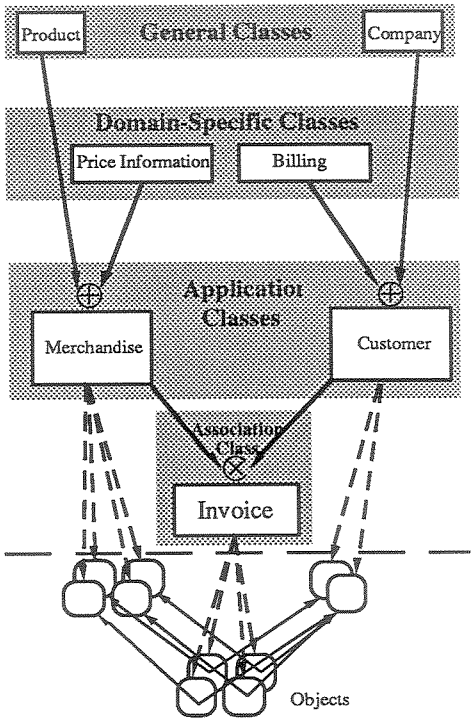


Figure 5.1b — AR System

In a library domain, library books are composed from a general book class and a domain-specific lending class. Similarly, library patrons are composed from a basic_person class and a domain-specific member class. Library books and patrons then participate in certain associations that define the nature of a library. A similar situation exists in an accounts receivables domain.

5.2.1 Library Book

Chapter 3 defined a general book class which can be composed with other information to produce new class definitions such as Library_Book. A book can be composed with a lending_item as shown in Figure 5.2:

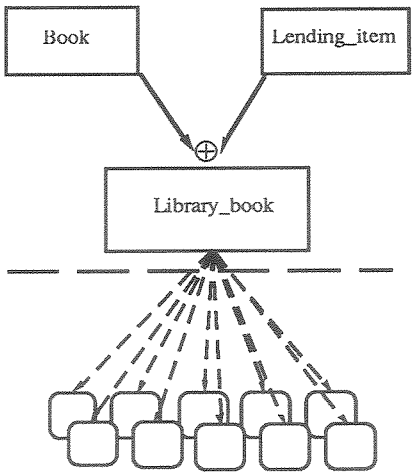


Figure 5.2 — Library_book: = (⊕ Book Item_to_be_Loaned)

Figure 5.3 gives a simple definition of a lending_item class that could be used for books, tapes, records, art, and other items that are to be loaned without charge.

```
Lending_Item = Class

    Loan_count :  Ratio_scale count in integer;
    Loan_status: Nominal_scale (loaned, not_loaned);
    Item_status: Nominal_scale (OK, missing, damaged);
    Acquisition_date : Interval_scale time in days;
    Cum_Loan_time :  Ratio_scale time in days;

OPERATIONS
    Inc_Cum_loan_time:  Ratio_scale time in days ((min 1)(max 90));
    Inc_Loan_Count: Constant time in days (1);
    Age:  Ratio_scale Time in days (derived Acquisition_date);
```

Figure 5.3 — Base Definition for a Lending_Item

```
Library_book  = Class

    Library_of_Congress (Name 1);
    Copy_number: Interval_scale ((min 1) (max 99) (Name 2));
    Title : Nominal Title_Type;
    Author : Nominal Person_Name_Type;
    ISBN : Nominal:  ID_Number;
    Copyright_date : Gregorian_date (min 1800);
    Cost : Ratio_scale money in dollars (gran .01) (min 1) (max 75);
    Weight : Ratio_scale weight in pounds (gran 1) (min 1) (max 35);
    Loan_count :  Ratio_scale count in integer;
    Loan_status: Nominal_scale (loaned, not_loaned);
    Item_status: Nominal_scale (OK, missing, damaged);
    Acquisition_date : Interval_scale time in days;
    Cum_Loan_time :  Ratio_scale time in days;

OPERATIONS
    Inc_Cum_loan_time:  Ratio_scale time in days ((min 1)(max 90));
    Inc_Loan_Count: Constant time in days (1).;
    Age:  Ratio_scale Time in days (derived Acquisition_date).;
```

Figure 5.4 — Book_in_the_library = (⊕ Book Lending_Item)

Note that a class may have properties that do not need to be mapped into the application. For example, book_weight is an attribute of a book that does not need to be known by the library, so it is left out.

5.2.2 Person

Another case where composition can be applied is when using and reusing the information required to build classes for common business purposes. Classes without naming attributes are generally created to be combined with other classes These classes contain information which, when added to a class with naming attributes, creates a new application class. Classes without naming attributes are useful mechanisms for storing domain specific information. Credit information, address information, rental information, and other domain specific and policy specific information can be neatly bundled up into these types of classes. These classes are common in other modeling techniques, and are sometimes called "mixins." One example of such a class is US_address which is shown in Figure 5.5:

```
US_Address = Class
    Address: US_street_address;
    City: US_City;
    State: US_State = Nominal_scale (AL, AK, AZ, AR, CA, CO, CT, DE, DC, FL,
            GA, HI, ID, IL, IN, IA, KS, KY, LA, ME, MD, MA, MI, MN, MS, MO, MT,
            NE, NJ, NV,NH, NM, NY, NC, ND, OH, OK, OR, PA, PR, RI, SC, SD,
            TN, TX, UT, VT, VA, WA, WV, WI, WY);
    Zip5 : Old_Zip_code;
    Zip4: New_Zip_code;
```

Figure 5.5 — US_Address

Figure 5.6 shows a creation_location class. Creation_location is a class without a naming attribute that can be used in a variety of different situations. When it is composed with a company_basic class (shown in Figure 5.7) a company class is created as shown in Figure 5.8. The composition process is indicated as follows:

Company Class : = (⊕ company_basic creation_location)

```
Creation_Location = Class
    Creation_date:  Gregorian_date;
    City_of_Creation:  US_City;
    State_of_creation:  US_State;
```

Figure 5.6 — Creation Location

```
Company_basic = Class

NAMING  ATTRIBUTES
    Name:  Company_name;
    Federal Tax_ID : Nominal_scale;
```

Figure 5.7 — Company_ basic

```
Company = Class

NAMING ATTRIBUTES
    Name: Company_name;
    Federal_Tax_Identification_number;

NON-NAMING ATTRIBUTES
    Incorporation_date: Julian_date;
    City_of_incorporation:  Nominal_scale;
    State_of_incorporation:  State_Type;
    Address: US_street_address;
    City: US_City;
    State: US_State
    Zip5 : Old_Zip_code;
    Zip4: New_Zip_code;
    Phone: phone_number;
```

Figure 5.8 — Company_Class: = (⊕ company_basic creation_location US_address)

The creation location can also be composed with a person to create a person class: person_class: = (⊕ person_basic creation_location). This operation is fully detailed in Appendix B.

5.2.3 Defining a Client

By organizing information into classes, the domain expert creates a system that allows different final classes to be created. For example, in Figure 5.9 a client is created from the classes: company, person, billing_info, and payment_info Note that the OR operator refers to the domain expert's selection of either a company or a person to be composed.

A client is a person or a company composed with billing and payment information. Client : = (⊕ (OR Company Person) Billing_Info Payment_Info). In a legal time and billing system, another class which could be composed to create a specialized client is shown in figure 5.13. The results of these compositions are shown in Appendix B.

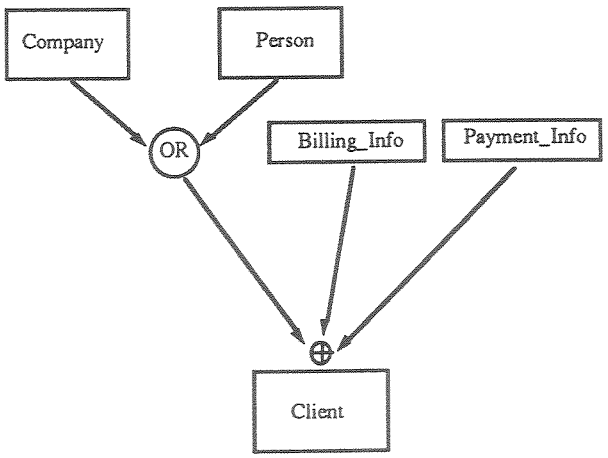


Figure 5.9 — Client : = (⊕ (OR Company Person) Billing_Info Payment_Info).

5.3 Composition With Conflicts

The previous composition examples have not had to deal with the case of naming conflicts between two attributes. This section examines the ability of the domain model to facilitate the composition of classes that have attributes which conflict or at least appear to conflict. The conflict problem was illustrated in the automobile example in Chapter 1. When composing an engine and chassis to create an automobile, conflicts appeared for both the age and weight attributes, Figure 5.10 shows this situation. In general, domain knowledge is required to resolve conflicts between the age and weight attributes.

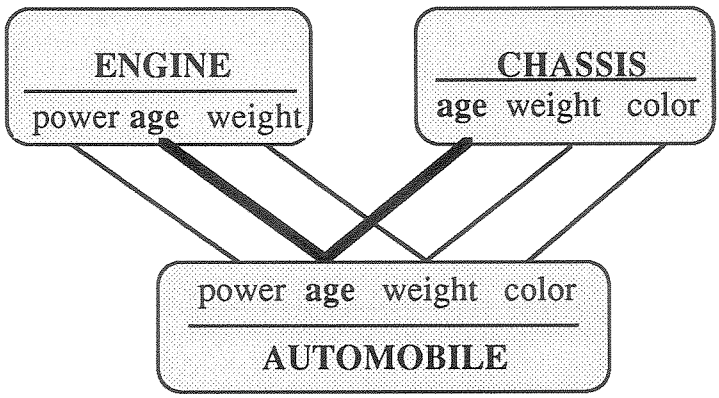


Figure 5.10 — Determining Age & Weight of an Automobile

Snyder [Snyder 88] describes three syntactic approaches to the conflict problem in multiple inheritance. Although these strategies provide work around solutions to the conflict problem, they fail to use any semantic information to resolve those conflicts. Consequently, the choice of a particular solutions is an ad hoc decision based on programming language design rather than domain semantics. Our approach uses domain knowledge to avoid a purely syntactic approach and consequently solves the problem for classes whose attributes contain the appropriate semantic information.

For example, weight is an attribute that can be composed using the plus operator that was defined in detail in Chapter Three. The plus algorithm can be summarized as follows: In order to combine two attributes with interval or ratio scales, say A1 and A2, into a new attribute A3: A1, A2, and A3 all must be of the same quantity , must have the same unit measure, and the measurement granularity of the unit of A3 must be at least as coarse as the most coarse of the measurement granularities of the units of A1 and A2.

Our definitions of attribute and class have also sidestepped another problem of classification that is sometimes seen in other forms of representation. For example, even for a well understood property like weight[1], objects in some domains do not always combine as cleanly as did the engine and chassis. For example, is the weight of a football team the weight of the biggest player? the smallest player? the average player? the list of all of the players? This problem can be approached in the model because weight, and all other attributes are defined in terms that capture semantics.

---

[1] The definition of weight in most domains is defined as an object's weight on earth rather than by the definition of mass times acceleration (with separate instantiations of average gravitational acceleration for each object). Although in most cases, this is considered common sense, in certain domain-specific applications such as space vehicle construction, this assumption is violated.

Figure 5.11 shows a solution in terms of a domain model.

```
Engine = Class
    Power : Ratio_scale power in horsepower;
    Weight : Ratio_scale weight in ounces;
    Age : Ratio_scale  distance in miles;
    Create_Date : Interval_scale time in days;

Chassis = Class
    Weight : Ratio_scale weight in pounds;
    Create_Date : Interval_scale time in days;
    Color : Nominal (red, yellow, green, blue);

Automobile := (⊕ Engine  Chassis)

Automobile = Class
    Power : Ratio_scale power in horsepower;
    Weight : Ratio_scale weight in pounds;
    Create_Date : Interval_scale time in days;
    Miles: Ratio_scale distance in miles;
    Color : Nominal (red, yellow, green, blue);
```

Figure 5.11— Scaled Version of Automobile

5.4 Relation to other paradigms

In a discussion of the underlying themes surrounding object-oriented systems, [Stefik 85] wrote that "...a map of the world is necessarily drawn from where one stands...." This comment seems to be especially true with the composition operator. Composition is a general purpose constructor that allows the domain expert to build complex classes from smaller previously existing classes. This definition is sufficiently general that it includes structuring techniques such as *part of* and *multiple inheritance* as special cases.

The automobile example illustrated composition that in other systems is sometimes known as a *part of* relationship. Both the engine and the chassis are parts of an automobile. This is a common special case of composition, and as has been shown in this section, leads directly to reuse of commonly occurring classes such as US_address.

The problem with a term such as *part of* is that while the nomenclature appears to convey very specific meaning, it is still subject to multiple interpretations. In the previous paragraph we stated that Figures 5.10, and 5.11 showed that an engine and chassis are related to an automobile by a *part of* link. But an equally valid realization of *Part of* is as a composition with no conflicts and no deletions. In the automobile example, this realization requires that both the age and weight of the engine, and the age and weight of the chassis, be maintained in the definition of an automobile.

Since composition is an operation that builds larger classes from smaller ones, it is capable of constructing either of these cases. Throughout this dissertation, we have attempted to build a methodology that encourages, and many times forces, domain experts

to define exactly what they mean by particular constructs. Since *part of* can be interpreted in several ways, we avoid its use in the methodology.

Multiple Inheritance is another term that has a variety of meanings in other systems. While all of these meanings have the effect of combining attributes from more than one class, interpretations differ as to the procedures for handling conflicts, and the addition and deletion of attributes. Using the definitions from our meta-model, the *multiple inheritance* (in most systems) violates the constraint that subclasses can consist only of attributes inherited or derived from their parent. Rather than using the ambiguous term *multiple inheritance,* we require the use of composition to specify the case sensitive meaning of the operation. Figure 5.12 presents an example that will clarify our view.

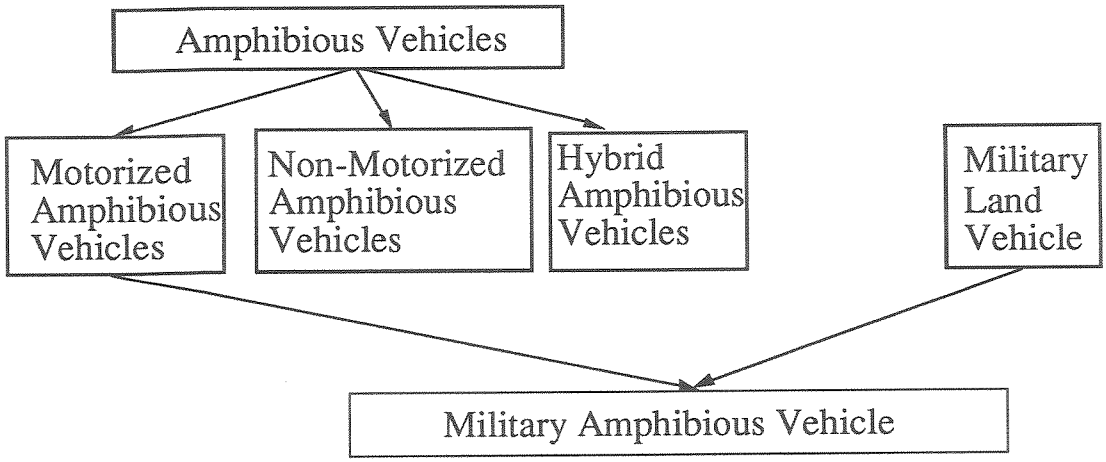


Figure 5.12 — Classic view of Multiple Inheritance

The figure shows a classic multiple inheritance example in which an amphibious vehicle is combined with a military vehicle to create a military amphibious vehicle. This combination is a reasonable interpretation at this level of abstraction, and in most systems, it is left up to the programmer in most to define what it really means to be a military amphibious vehicle. Our definition of specialization precludes this interpretation.

Figure 5.13 illustrates the same situation as modeled with our methodology. At this level, composition is once again used to combine attributes, and can be viewed as establishing more *part of* relationships, or of simply performing multiple inheritance as it is pictured in Figure 5.12.

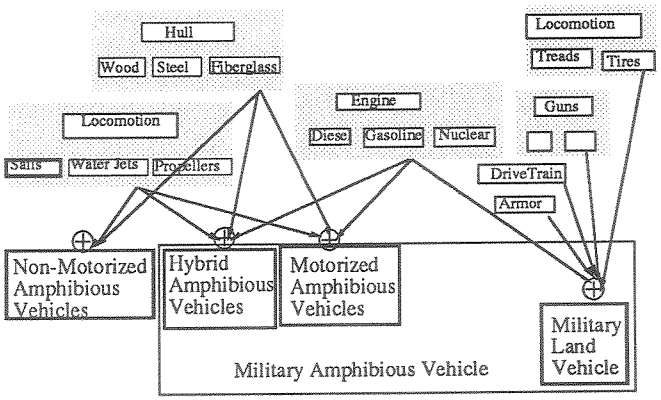


Figure 5.13 — Composition Within A Domain Model

Chapter 6 – Association

6.1 Introduction

Association is the process of creating a new class that establishes a relationship between at least two other classes by taking the union of their naming attributes and using domain knowledge to add new attributes as needed. The association operator can be viewed in several different ways. At one level of abstraction, it indicates the dynamic nature of a system. However, from a strictly definitional standpoint, it can be viewed as a composition operator in which there are guaranteed to be no name conflicts (because this is a constraint enforced by the model) and all attributes except for naming attributes are eliminated.

The prefix symbol ⊗ is used to indicate the association process as in the expression: $C_M := (\otimes\ C_1\ C_2 \ldots C_n)$. The following definition gives the algorithm for the association process:

C_M is a class that is the association of the classes $C_1\ C_2 \ldots C_n$. The algorithm is as follows:

$C_M := (\otimes\ C_1\ C_2 \ldots C_n)$.

1. $\mathcal{A}(C_M) := \mathcal{N}(C_1) \cup \mathcal{N}(C_2) \cup \ldots \mathcal{N}(C_n))$
2. $\mathcal{A}(C_M) := \mathcal{A}(C_M) \cup$ selected_new_attributes$_{\text{✍}}$(attribute_library)

Definition 6.1— Association

The checkout class creates objects that are *linked* at run-time to the instantiated objects of the library book class and library patron class. The association operator can be used to create an association class from classes with naming attributes. Associations are classes that capture the semantics of the relationship between two or more other classes. They are called associations because they contain naming attributes from two or more external classes, and they associate the objects pointed to by those naming attributes as shown in figure 6.1, and in the center of the figure 5.1a & b where the association operator, ⊗, creates a checkout class that associates the library book and library patron.

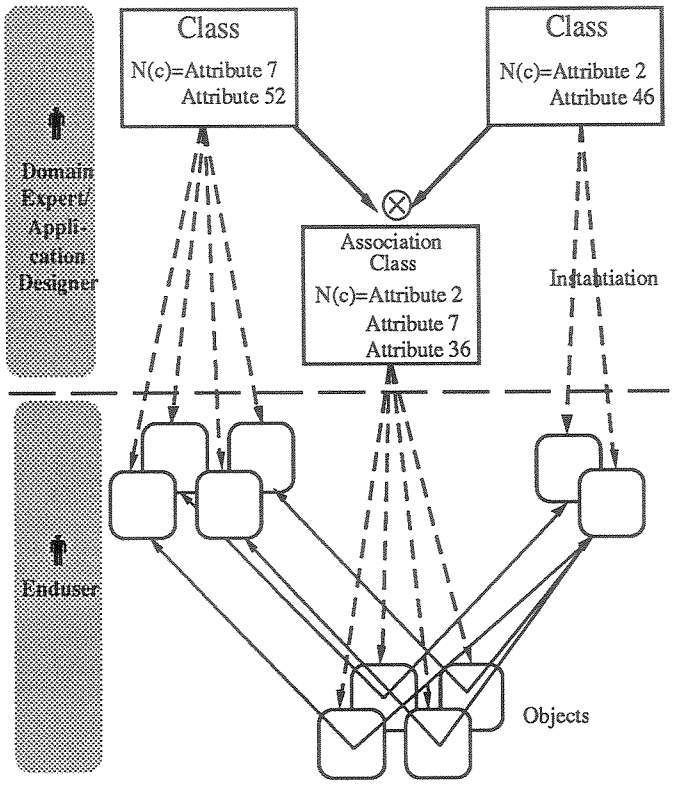


Figure 6.1 — Associations

As was seen in the library example (Chapter 1, Figure 1.3a), when a book is checked out by a person, the check-out class instantiates an object which records the association between the relevant person and book object. The domain model goes well beyond capturing the surface semantics represented by more traditional approaches such as the entity relationship model [Chen 76][1] . The classic ER model, in addition to overloading the term relationship with a variety of IS-A meanings, captures only the cardinality of object association. Mapping the ER model to our modeling methodology, one can see that dependent entities are roughly equivalent to classes without naming attributes. Within a domain model, classes capture a variety of additional information not typically captured by data base schemas.

6.2 Association as Contract

One useful way to view the modeling methodology's treatment of an association between two classes is to view that relationship as an implicit agreement or contract between those classes. The role of the modeling methodology is to allow the domain expert to specify that agreement or that contract in terms of the domain model.

Consider some of the functions of the class that associates person and book for the check-out operation. In this case, the object represents an agreement by a library patron to return a book within a certain period of time. In the event that the book is not returned, the object which captures the semantics of that agreement is responsible for a series of events which will enforce the agreement.

---

[1]See also related work Entity Relationship Models

When a book is first checked out, the check-out class does relatively little. It records the fact that the book is checked out by a particular person by creating an object that associates the person_object with the book_object, and it invokes an operation on the person_object which increments the count of the current number of books checked out by that person, and invokes an operation on the book_object which marks the book that was checked out as no longer available.

If the person returns the book within the normal time allotted for its return, then the object instantiated by the check-out class is deleted by the delete operation for that class; the number of books checked out by the person is decremented and the book is marked as once again available for check-out.

However, consider the case where the book is not returned within the allotted time period. In this case, the check-out object is responsible for initiating actions to get the user to return the book. Depending upon the policies and protocol of the library, the object might produce an overdue notice or set of notices, followed by a reminder letter followed by a last notice for return, followed by a statement informing the user that the book was lost. Also depending upon the policies of the library, it might at some point change the status of the person in the library from able to check out books to not able to check out books. Similarly, it might change the status of the book from checked_out to lost or stolen. These events and their actions are illustrated in Figure 6.2.

Event	Time	Class Action
End of day processing	< n	none
n		Issue Overdue notice
x		Issue 2nd Notice
y		Issue Final Notice
z		declare book stolen
Return	<n	None
	n<Time<z	Calculate fine - assign to person
	Time >= z	Issue letter - Too late to return book

Figure 6.2 — Library Events

While the terms of agreements, and the policies for enforcement of these agreements, may differ radically from program to program, the principles are the same. In the case of an accounts receivable system, a person or a company receives merchandise and also must pay for it within a certain period of time. The semantics of the enforcement of that payment are the policies of a company with regards to accounts receivable. For example, the statement *"our terms are 2/10 net 30"* means that the purchaser is given a 2% discount if an item is paid for within 10 days. In the event that the item is not paid for in 10 days the purchaser is supposed to pay for the item within 30 days. In a more complicated case, additional interest may begin to accrue after the 30 days have passed. These accounts receivable terms are illustrated in Figure 6.3:

Event	Time	Class Action
End of day processing	< =10	Amt_due := total- total * discount
	10<Time<30	Amt_due := total
		Send notice
		Send 60 day notice
		send 90 day notice
Payment Made	< =10	Amt_due := total- total * discount
	10<Time<30	Amt_due := total
	Time >= 30	Amt_due:= total + total*

Figure 6.3 — AR Relationship Events

Regardless of the actual terms and conditions, the principle is the same. Relationship classes capture the semantics of agreements, be they explicit or implicit, between other classes.

6.3 Semantics of Instantiation & Deletion

In chapter 3, instantiation and deletion were introduced as paired operations that were attached to all classes that created objects. Association classes create objects, but must also maintain the cardinality of the links to other objects, as well as maintaining link integrity in the event that a linked object is deleted.

```
Checkout = Class
        Date_Checked_out Gregorian_date

    EXTERNAL
        CLASS Person (NAME Person_Name)(CARD M))
        CLASS Book (NAME Call_Number)(CARD 1))

    OPERATION
        Date_due := check_out_length(book.type, person.type) + current_date
```

Figure 6.4 — Checkout Class

```
Invoice = Class
        Date_Purchased Gregorian_date
        Sub_total_Purchase: ARDollar
        Discount_P:  Percent_Money-in_Dollars (min 0) (max .40)
        Interest_P: Ratio_Scale_Money-in_Dollars_Modifier (min 0) (max .22)
    DERIVED
        Discount_period: Ratio_Scale Time in days (min 0) (max 180)
        Tax := Sub_total * Company.taxrate
        Total_Due := Sub_total + Tax
    EXTERNAL
        CLASS Customer (NAME Customer_Name)(CARD 1))
        CLASS Item (NAME Item_ID)(CARD M))

OPERATION
        Date_due := check_out_length(book.type, person.type) + current_date
```

Figure 6.5 — Invoice Class

6.4 N-ary associations

In the examples in this chapter, all of the associations have been binary associations. However, it is sometimes the case that an N-ary association should be considered. For example, parts, manufacturers, and suppliers is a case where a three-part relationship is a common semantic interpretation. While N-ary associations can always be constructed as compositions of binary associations, this composition process can obscure the semantics of the high level application domain association. Our approach is similar to Kent's idea of two different views of the arity of associations [Kent 78] Internally, composition is regarded as a binary operator, but to the domain expert, composition is regarded as an N-ary operator.

6.5 Transformational Implementation Issues

At an implementation level, all classes map cleanly into tables where each attribute represents one column and each object one row. Viewed from the standpoint of classic relational data models, the set of attributes defines a relation, and the class contains the integrity constraints that determine the creation of new tuples (rows).

Although domain models can always be implemented in this manner, efficiency considerations dictate that refinements on this scheme occasionally be performed. In the case of primitive classes (e.g. ones that have not been constructed from composition or association) that create objects, few options exist besides determining whether or not to build an index on a particular attribute or set of attributes.

In the case of classes created by the association operator, the options are also quite limited. Essentially, these classes always include the naming attributes of the classes they are associating, as well as any additional attributes that are required to add meaning to the association relationship. Although the association options are limited, a spectrum of options exist for classes that are created by the composition operator. These

implementation options range from irreversible binding to reversible binding of the attributes.

Totally reversible binding refers to attribute binding that is accomplished by creating a table that uses naming attributes as pointers that allow indirect access of the attributes in the classes named by the naming attributes. In contrast, irreversible binding refers to a composed class implemented by directly storing all of the attributes in the same table. Irreversible binding allows more efficient access to the attribute values then the traversal of pointer chains required by reversible binding. Although irreversible binding is inherently more efficient, it must be prohibited in cases where it would cause update and deletion anomalies (as described in the discussion of normalization in Chapter Three).

Chapter 7 – Related Work

7.1. Introduction

In this dissertation, we have outlined a domain meta-model and a methodology for its instantiation into specific domain models. In the introduction to this dissertation, we pointed out that while programming is well understood enough to model and teach, application domain knowledge is rarely explicitly modeled. In this chapter, we review other modeling efforts that are related to our own.

Domain models are representations of an application domain that can be used for a variety of operational goals in support of specific software engineering tasks or processes. Operational goals are always implicit in the construction of a model and are essential to understanding the form and content of a model that captures domain knowledge. Within software engineering these goals include:

1. understanding and analyzing application areas (domain analysis).
2. eliciting and formalizing software requirements and specifications (requirements/specifications).
3. providing for new software engineering paradigms (methodologies).
4. assisting in code development and maintenance (system evolution).
5. capturing and communicating design decisions and rationales (communication).
6. identifying semantics of existing code (reverse engineering).
7. resolving design decisions (decision modeling).
8. training designers and end users (education).

Our research has been focused on developing a meta-model and methodology that facilitates achieving the first two goals listed above. In doing this we have also made progress towards goals 3 and 4.

Domain modeling is a field that draws from research in design methodologies, database schema, object-oriented programming, and knowledge representation. The next sections review the antecedents of domain modeling that most effected our research. These reviews are terse and only intended to review some of the salient high points of these broad research areas. Following this review, a more targeted review of domain modeling and program transformation is presented in terms of the origins and genealogies of domain modeling research. Finally, the results of our research are presented as an incremental improvement upon current modeling and suggestions for future research are detailed.

7.1.1. Object Oriented Design and Programming

SIMULA [Dahl 66], [Birtwhistle 73] is generally recognized as the first object-oriented language. Written as an extension of ALGOL 60, this SIMUlation LAnguage was designed to provide a set of basic building blocks to map discrete event simulation problems into programs. By adding structuring primitives called classes, [Dahl 72] created a representation that provided a convenient way to view and program a variety of classical programming problems.

Object-oriented programming added two new dimensions to the tradeoffs that have to be made in programming languages. Figure 5.1 shows inheritance and encapsulation as new issues in programming.

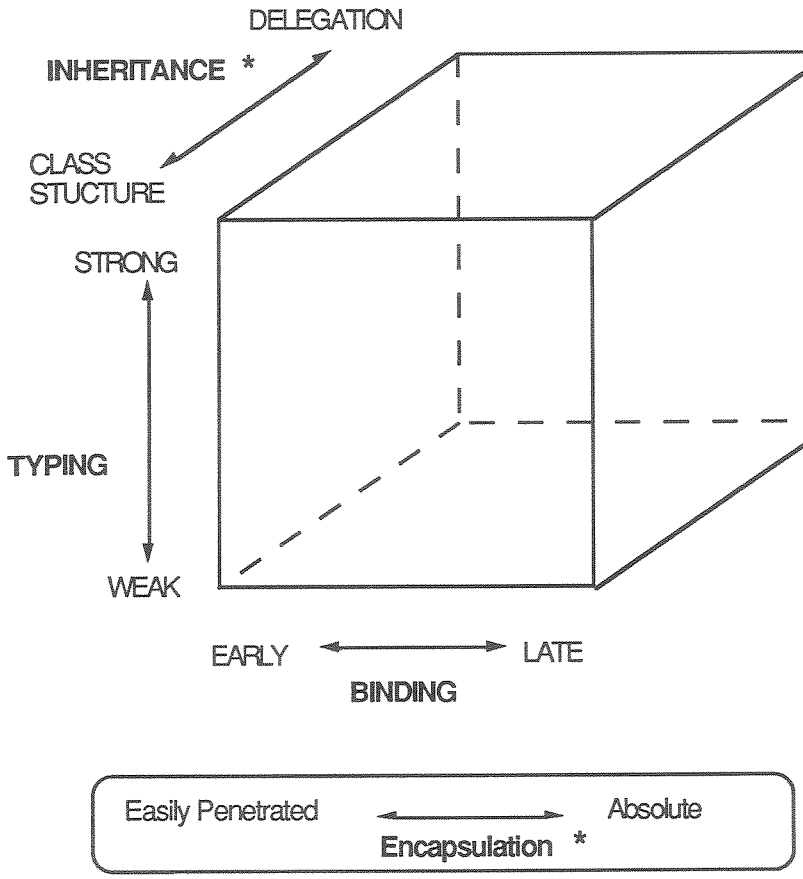


Figure 7.1 — Some Considerations in (Object-Oriented)* Programming

Hydra is another language important in the history of object-oriented programming because it was the first major operating system project to use the concept of a class to express the abstractions inherent in the design and implementation of the kernel of an operating system. The authors of Hydra decided to approach structuring from the concept of a *class* as had been defined in SIMULA, and extended to monitors [Hoare 74]. [Wulf 74] and [Wulf 75] describe the research as it was in progress, while [Wulf 81] is both a summary and a retrospective view of the project after its completion. Further work using objects as a structuring device for a vertically partitioned operating system can be found in [Browne 82] and [Browne 84].

Smalltalk was created as a completely object-oriented programming and execution environment. First described by [Ingalls 78], the system began as Smalltalk-76 and was rewritten to become Smalltalk-80. A guide to the language is provided by [Goldberg 83] and a description of the programming and run-time environment is given in [Goldberg 84]. As was the case in the previously discussed systems, Smalltalk is based on the concept of an object that stores both *instance variables* (state) and *methods* that alter the state, send messages, or interact with the user.

CommonLoops [Bobrow 86] and its predecessor Loops and [Stefik 85] extend the function calls of Lisp to an object-oriented model that is designed to suit the needs of AI researchers. Just as Simula was created to provide additional features for use with ALGOL 60, CommonLoops adds its own set of features to Common Lisp.

In addition to supporting class and hierarchical inheritance structures, CommonLoops also supports *multiple inheritance,* which is a technique for inheriting the union of the instance variables and methods from more than one class or superclass. The structure resulting from multiple inheritance is an inheritance graph instead of a tree. By a careful construction of the graph, syntactic precedence relations can be provided by the system designer. Although multiple inheritance can be used within Smalltalk [Borning 82] and [Ingalls 86], it has been institutionalized in LOOPS and Flavors [Moon 86]. Further discussions of inheritance are found in [Lieberman 81], [Meyer 86], and [Snyder 86].

Comparing our research to *standard* object-oriented research is difficult, because object-oriented programming researchers have introduced a myriad of definitions for terms such as inheritance, delegation, class, object, superclass, subclass and so on. As [Stefik 85] notes in his survey article on object oriented programming, a map of the world is best drawn from where one stands. Many of the definitions depend upon the viewpoint of the researcher who designs a system. However, there does seem to be some commonality in, at least, the definition of an object as paraphrased from ([Dahl 66], [Goldberg 83], [Stefik 85], [Booch 86], [Cox 86], and [Meyer 88]).

> *An object, in an object-oriented model, is an abstraction of an entity that is characterized by its state and a set of operations that access or change that state. State is defined in terms of the properties or attributes of an object.*

Chapter 2 defined our notion of attribute which extends the semantically bare data types that are used even within object-oriented programming. Our definitions for attribute, class, object, superclass, subclass, and so on are summarized in Appendix D.

7.1.2. Data Base Schemas

Data base schema design is closely related to certain aspects of domain modeling. Data models in general, and data base schemas in particular, attempt to capture semantics of the underlying world. The reasons for doing this are simple. By capturing the semantics of an underlying domain, data base schemas can be used to design data bases that can be used efficiently whose *integrity* can be maintained. By integrity, data modelers normally mean a set of constraints, axioms, relationships, or specifications that remain invariant throughout the lifetime of the data stored in the data base.

There are many semantic data models which are surveyed by [Peckham 88], [Tsichritzis 82], and [Hull 87]. *Entity Relationship (ER) modeling* was described in a seminal paper by [Chen 76], and concepts of generalization and aggregation were added by [Smith 77]. ER models divide the world into entities, which are things that exist in the world, and relationships, which are the ways those entities are related. ER modeling is related to our work because it is one reasonable approach to the general problem of conceptual modeling [Brodie 84]. In ER modeling there is often a question as to the distinction between entities and relationships. In our model, as was shown in chapter 4 in

the section on association classes, this problem is addressed by using domain-specific knowledge to guide the domain expert to make the appropriate classifications.

7.1.3. Design Methodologies

There are a variety of design methodologies that are used to create application programs. Although they are not directly relevant to our work, certain methodologies can also be considered to be modelling techniques, as in an analysis [Bruns 88] that included Jackson System Development (JSD) [Jackson 83] and Booch's Ada design method [Booch 86] as domain modeling systems.

Jackson Structured Design (JSD) is one of the few methodologies [Jackson 83] that clearly describes techniques for eliciting the objects and operations for a domain. While certain of Jackson's views differ from ours, his approach is related to our own from the standpoint of using a domain expert to try to model system behavior. His approach is different from ours in his description of entities (classes in our model) for which he states that:

> *A JSD entity must: 1) perform or suffer actions in a significant time-ordering, 2) exist in the real world and not merely within the system, 3) be capable of being regarded as an individual, and if there is more than one entity of a type, of being uniquely named.*

Two types of entities are considered to exist within our model. There are those entities that have an existence outside of the (not necessarily computerized) application system, and those that have an existence only by virtue of the existence of the application system.

Yourdon [Yourdon 79] and SADT [Ross 77] are techniques commonly used in industrial practice. Both of these have a waterfall model orientation. Shlaer [Shlaer 88] describes a methodology, within a relational database model, for domain analysis.

7.1.4. Knowledge Representation KL_ONE

Knowledge representation is a large field with operational goals that range from restricted (such as application domain modeling) all the way to Doug Lenat's Cyc [Lenat 88A], [Lenat 88B], [Lenat 89], where the eventual goal is to represent all world knowledge. Summaries of knowledge representation that most closely relate to our own research are collected in [Brachman 85], [Brodie 86], and [Brodie 84], with a summary paper by [Mylopoulous 84].

The most direct influence on our work from this area has come from Brachman in his clear presentation of *IS-A* links [Brachman 83], his presentation of the importance of definitional information [Brachman 85b], and the clean separation of concept from implementation in his KL_ONE language [Brachman 85a] which is a general knowledge representation system used in artificial intelligence programs.

Our goal has been to establish a system with a clear definitional framework that allows succinct and unambiguous statements by domain experts about application domains.

7.2. Domain Modeling

The field of domain modeling is young. In September of 1988 and October of 1989, we organized workshops whose intent was to exchange ideas and arrive at a consensus on

domain modeling issues, terms, definitions, and measures of validation. Although progress has been made, there do not yet appear to be consensus views on all of these items. This section begins by presenting general approaches to domain modeling from other research perspectives, and then narrows in on the development of domain modeling and automated programming.

One difficulty in discussing domain modeling is that researchers have differing views on the term *model*. Throughout this dissertation we have used the terms meta-model, domain model, and modeling methodology. Figure 1.1 illustrated our view of these terms. The meta-model consists of a set of primitives that can be instantiated to produce a domain model within a particular domain. The modeling methodology is the methodology that a domain expert uses to instantiate a particular domain model using attributes, classes, inheritance, and composition.

7.2.1. Knowledge-Based Transformation Systems

Knowledge-Based Transformation Systems use information about a domain to achieve good mappings when algorithmic methods are not yet well understood, don't exist, or are computationally intractable. Most existing knowledge-based transformation systems operate in what we are calling *computer science and language domains* (the middle two boxes of Figure 1.1 of Chapter 1). This is a critical part of the overall solution paradigm but is not part of this research.

7.2.2. GIST

GIST represents a fifteen-year effort directed by Bob Balzer [Balzer 85] at the Information Sciences Institute (ISI) to produce a knowledge-based system for transforming program specifications to executable code. Because GIST has been developed over a number of years, there is no single definitive report that outlines all of the features or even the operational goals of the project. The philosophy at ISI has been to develop GIST interactively in order to test their evolving ideas about program specification. Further write-ups are available in [Balzer 78], [Balzer 81], [Balzer 79], [Balzer 76], [Feather 87A], [Feather 87B], [Feather 89], [Wile 83], and [Bruns 86a]. (See also Glitter, a transformational system by Fickas Glitter was developed by [Fickas 85])

GIST provides many different language features for modeling domains. Essentially, a GIST model has objects with both single and multiple inheritance. These objects are related to each other through relations. GIST also has demons, global constraints, and detailed methods for state transitions.

7.2.3. PSI, PECOS, LIBRA, CHI

The PSI project (described in [Green 76], [Green 78], [Barstow 79A], [Barstow 79B], [Kant 81A], [Kant 81B], and [Kant 83]) used a knowledge-based approach to synthesize programs within the domain of computer science. The PECOS component of PSI stored knowledge about the domain in terms of refinement rules. LIBRA helped create efficient programs by using its knowledge about costs of computation to limit the number of refinement rule expansions needed to create a program.

CHI [Green 82], [Westfold 84] was the successor to PSI, and used an internal representation of objects that served to structure the knowledge base. This structuring of

knowledge helped organize the knowledge base. CHI also differed from PSI in its approach to selection of transformations; it used user supplied information as opposed to the LIBRA style of cost calculation.

CHI's successor is REFINE [Smith 85] , a currently available commercial system that is used in a variety of environments.

7.2.4. ΦNIX

ΦNIX [Barstow 89] is an automated programming system operating in the domain of oil well logging tools. Barstow has strongly influenced our work by his long-held belief that automatic programming systems must be domain-specific, and that domain specific knowledge should be modeled in a form available to the program [Barstow 84]. The major effort in the implementation of ΦNIX has been directed at the transformational system as opposed to the specification system.

7.2.5. Requirements Apprentice

The Requirements Apprentice [Reubenstein 89], [Rich 87] is an offshoot of the MIT Programmers Apprentice project which is further described in [Rich 88B], [Rich 78], [Rich 82], [Waters 82], and [Waters 85]. The overall goal of the Programmers Apprentice project is to provide assistance to programmers by freeing them of some of the administrative details in programming, and it is only tangentially related to our work.

Reubenstein's view of the requirements of an apprentice is as an assistant to an analyst, who then interacts with a user/client. In this respect, his approach differs from ours in its concentration on a system for analysts. However, his focus is on the development of a cliche library that can be viewed in terms of a domain model. Furthermore, he is "developing a theory of coherent requirements, requirements descriptions and [identifying and codifying] reusable components and software requirements."

7.2.6. IDeA, ROSE

The Microelectronics and Computer Technology Corporation (MCC) has several researchers who either are or have been involved with domain modeling. Most closely related to our research is the work of Lubars on IDea [Lubars 88]and ROSE [Lubars 89]. Lubars is concerned with differing views of abstract design schema and is targeting his work towards programming-in-the-large type designs. This work is an extension of his dissertation research [Lubars 86], [Lubars 87a].

Reverse engineering is the current thrust of Biggerstaff's research [Biggerstaff 87], [Biggerstaff 89A], ,[Biggerstaff 89B], which is not directly related to our work.

Also of interest are MCC's analysis efforts. In one report, [Bruns 88] Bruns and Potts analyzed five different approaches to domain modeling.

7.2.7. KATE, ASAP, AHS

At the University of Oregon, Steve Fickas [Fickas 87] has developed several projects that combine artificial intelligence techniques with the software specification process. These projects include KATE [Fickas 87], a project whose goal is to automate aspects of the software specification process, ASAP [Anderson 89], and [Robinson 89]. Working

with the domain of conferencing, Fickas is building an interactive system that can help the user analyze their requirements and produce a reasonable specification. One way to view KATE is as an automated systems analyst.

Our goals are similar, but our approach is substantively different, from that of the KATE project. While Fickas concentrates on domains, such as conferencing, that are not well understood, even by domain experts, we are concentrating on domains for which a domain expert has a clear conceptual model. Furthermore, the type of advice-giving which is attempted by KATE is not one of our operational goals.

ASAP (automated specifier and planner) is a program that uses planning techniques within the area of specification design. ASAP assists designers by working with operators and plans in an attempt to help users achieve desirable goals and prevent undesirable events. ASAP's goals are high-level ones which attempt to "find a productive division of labor between a system and a user according to the ability of each."

Robinson's work [Robinson 89] attempts to integrate specification design with multiple perspectives of a system. He is formalizing Feather's transformational implementation paradigm [Balzer 81]. He has created a system called AHS which maintains a domain model of goal perspectives. These goal perspectives can be elaborated by applying a sequence of specification editing commands which, in effect, operationalize the perspectives. Assuming the two specifications are in the AHS environment, the specification components can be merged by applying in sequence: correspondence identification, conflict detection characterization, conflict resolution, and resolution implementation.

7.2.8. RML

RML (Requirements Modelling Language) was developed by Greenspan to create "a knowledge representation approach to software requirements definition." His dissertation is [Greenspan 84]; an earlier description of RML is [Greenspan 82], and a subsequent write up is [Greenspan 86].

He approached the problem from the standpoint of knowledge representation. A model in RML consists of entities, activities, and assertions for representing concepts, which it organizes through aggregation, classification, and generalization. An RML model is defined in terms of a translation to first-order logic. He also notes that SADT (Structured Analysis Design Technique [Ross 1977] can be used as an instantiation methodology. RML was developed as part of the Taxis project at the University of Toronto [Mylopoulous 84], and is based on the same framework as the Taxis language. However, Greenspan did not complete RML from the standpoint of operationalizing the definitions.

7.2.9. UCI — DRACO

The term *domain analysis* seems to have been coined first by Jim Neighbors in describing one aspect of Draco which he constructed as part of his dissertation research [Neighbors 80]. (Draco is summarized in [Neighbors 84a] and also described by the manual [Neighbors 84b].)

Within the context of Draco, the term *domain* refers to more than just application domains. In the terms of Figure 1.1 of Chapter 1, Draco would be used to create a different domain model for each horizontal bar in the figure. Consequently, one way to

view Draco is as a meta-system in which all of the domains required to map from requirements to implementation are able to be defined. In actual practice, Draco has been used to port itself from UCI Lisp to Franz Lisp [Arango 89], as well as to solve a data base problem in Neighbor's dissertation.

Domain models within Draco are language based. Each Draco domain consists of a domain-specific language and a prettyprinter for that language. The language is specified in a modified BNF notation, that, with backtracking extensions, allows context-sensitive languages to be defined. Objects and operations within the domain are defined in terms of a set of refinements which allow mappings from one domain to the next. Transformations, on the other hand, are designed to map between objects and perform optimizations within a particular domain.

In Neighbor's view, domain analysis is the process of developing a domain specific language within the constraints of the Draco paradigm. Neighbors provides no methodology for this design, but this issue has been addressed by Arango in his dissertation work which is summarized in [Arango 89]. Arango starts with Neighbor's formulation that

> *A domain analysis is an attempt to identify the objects, operations, and relationships between what [that] domain experts perceive to be important about the domain.*

He then points out that Neighbor's definition is ambiguous, not operational, and failed to specify a validation criterion. He further argues

> *Against the possibility of practical procedures for capturing the 'true' ontology and semantics of arbitrary problem domains. If they existed we would have succeeded in formalizing the scientific method.*

Arango's dissertation is a presentation of domain analysis in terms of a model of reuse. Arango points out the danger of comparing dissimilar techniques, and concludes that:

> *'Pure' domain analysis strives for the synthesis of (anticipatory!) theories about domains of reality. This task has eluded systemization for centuries and nothing suggests that effective procedures will be developed in the near future.*

While the context of Arango's concerns are within the overall analysis of all domains within Draco, we believe that our research in the area of well-understood, small, transaction-processing environments has shown that the conceptual models of domain experts can be codified in a systematic and formal manner.

7.3. Previous Research

In the long run, we believe the automated programming paradigm shown in figure 5.4 is the right way to conceptualize software development. At the lowest level of the system (Figure 5.4), a type manager will implement the composition of primitive functions that are the actual application program. In [Iscoe 86], we described CRTForm, a meta-system that produced type managers that implemented a set of primitive operations for forms on fields. Although that is not the emphasis of this research, it would be a critical part of large systems of this type. Other kernel implementations of what we would consider type

managers are found in [Almes 83], [Deutsch 83A & 83B], [McCullough 83], [Pollack 81], and [Wulf 74].

For the purposes of this research, we assume that it is possible to produce a transformational system that produces executable output. An example of this in our previous work is ACS [Iscoe 88a], a transformational system operating in the domain of relational data base systems.

Chapter 8 - Conclusion

8.1. Results

This dissertation has defined a meta-model for application domain knowledge and described a methodology for its instantiation by domain experts into domain-specific models. The emphasis is on domain characterization techniques that could be used to instantiate different domain models.

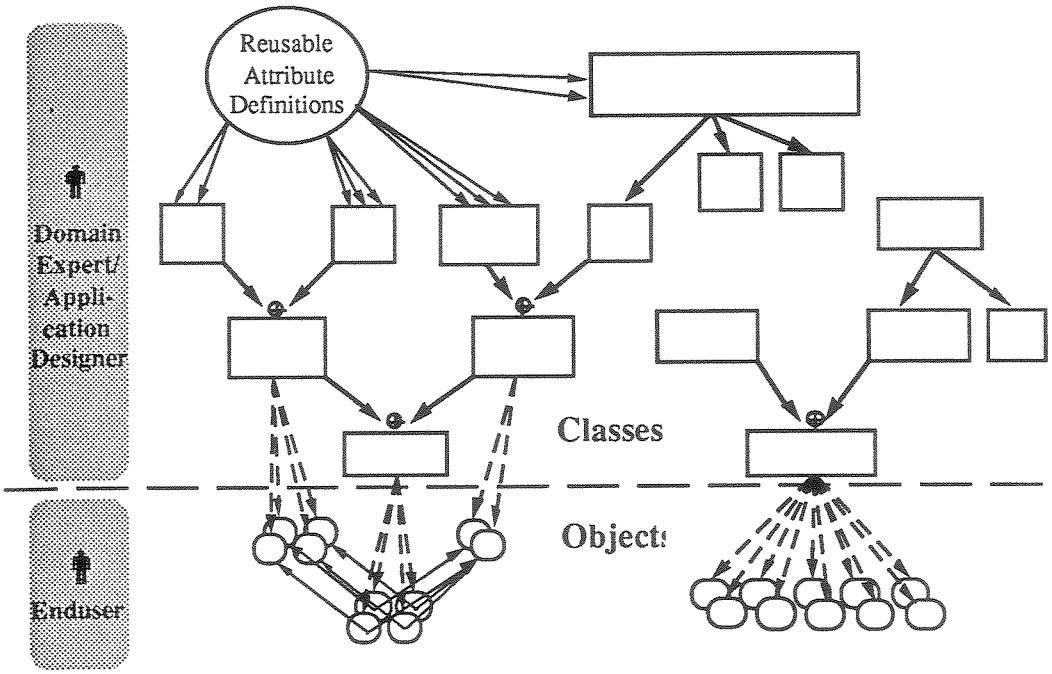


Figure 8.1 — Overview of the Model

Standard computer data types are semantically bare methods of representation that are unable to capture fundamental concepts in an application domain. Attributes are meta-model primitives that capture the semantics of domain properties in terms of scales (from mathematical measurement theory), units, quantities, granularity (from the physical sciences), population parameters (from statistics), and value set transitions.

Classes encapsulate sets of attributes, provide for the definition of derived attributes, allow additional operations to be defined, are responsible for object instantiation and deletion, provide for the definition of axioms, and maintain summary statistics on object sets.

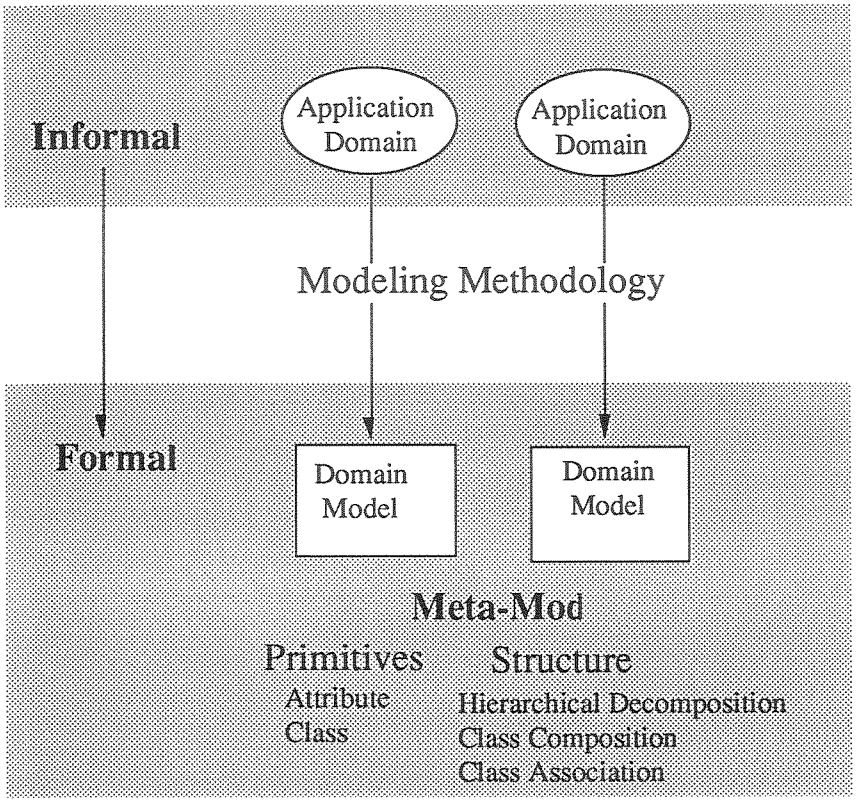


Figure 8.2 — Instantiating Domain Models

Domain models are instantiated by representing specific application domains in terms of attributes and classes. Classes are organized and structured using hierarchical decomposition to create subclasses, and composition and association to construct larger classes from smaller ones. Hierarchical decomposition is the process of developing a class hierarchy by using attribute restriction to specialize a class into subclasses. Composition is the process of creating a new class from two or more other classes by taking the union of their sets of attributes and using domain knowledge to resolve conflicts, eliminate unnecessary attributes, and add new attributes as needed. Finally, association is the process of creating a new class that establishes a relationship between at least two other classes by taking the union of their naming attributes and using domain knowledge to add new attributes as needed.

Examples were given in terms of library, accounts receivable, and other related problems.

8.2. Future Research

This dissertation has presented a domain modeling technique that generalizes across programming-in-the-small transaction-oriented business application domains. Our emphasis has been on the meta-model and the methodology for instantiation of that model into domain models.

In the long term, we believe that results from this dissertation, and others like it, will eventually allow designers, who are neither computer programmers nor domain experts, to construct application programs by declaratively describing and refining specifications for

the programs that they wish to construct. Our eventual goal is achieve the paradigm shown in figure 8.3.

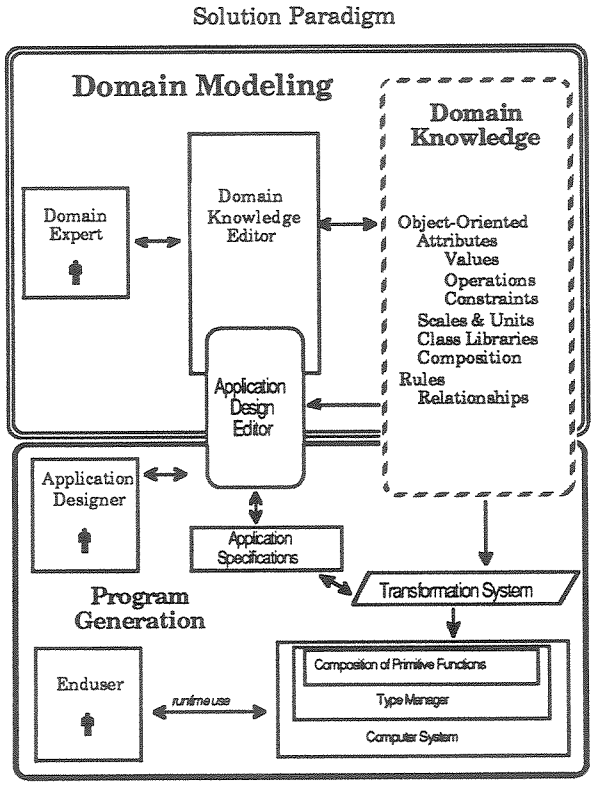


Figure 8.3 — Future Research

There are many possible paths to take with this research. Four areas we are investigating are:

- Applying attribute characterization techniques to certain aspects of programming-in-the-large problems.
- Implementing the transformational code generation section of the solution paradigm.
- Testing the meta-model against larger and more complex examples in order to better understand domain boundaries.
- Developing a subtraction analogue for composition.

Appendix A – Library Example

The library example is a classic software engineering problem that was introduced by Susan Gerhart and then published by [Kemmerer 1985] who used it a sample problem to illustrate a specification language. It was used as a sample problem in the Fourth Annual Workshop on Specification and Design [IEEE 1987] which resulted in twelve separate papers each detailing their approach and solution. [Wing 1988], an author of one of the twelve papers, gives a summary of the solutions presented at the conference. Another version of a library problem is found in [Jackson 1983]. The problem as presented in [IEEE 87] was the following:

Consider a small library database with the following transactions:

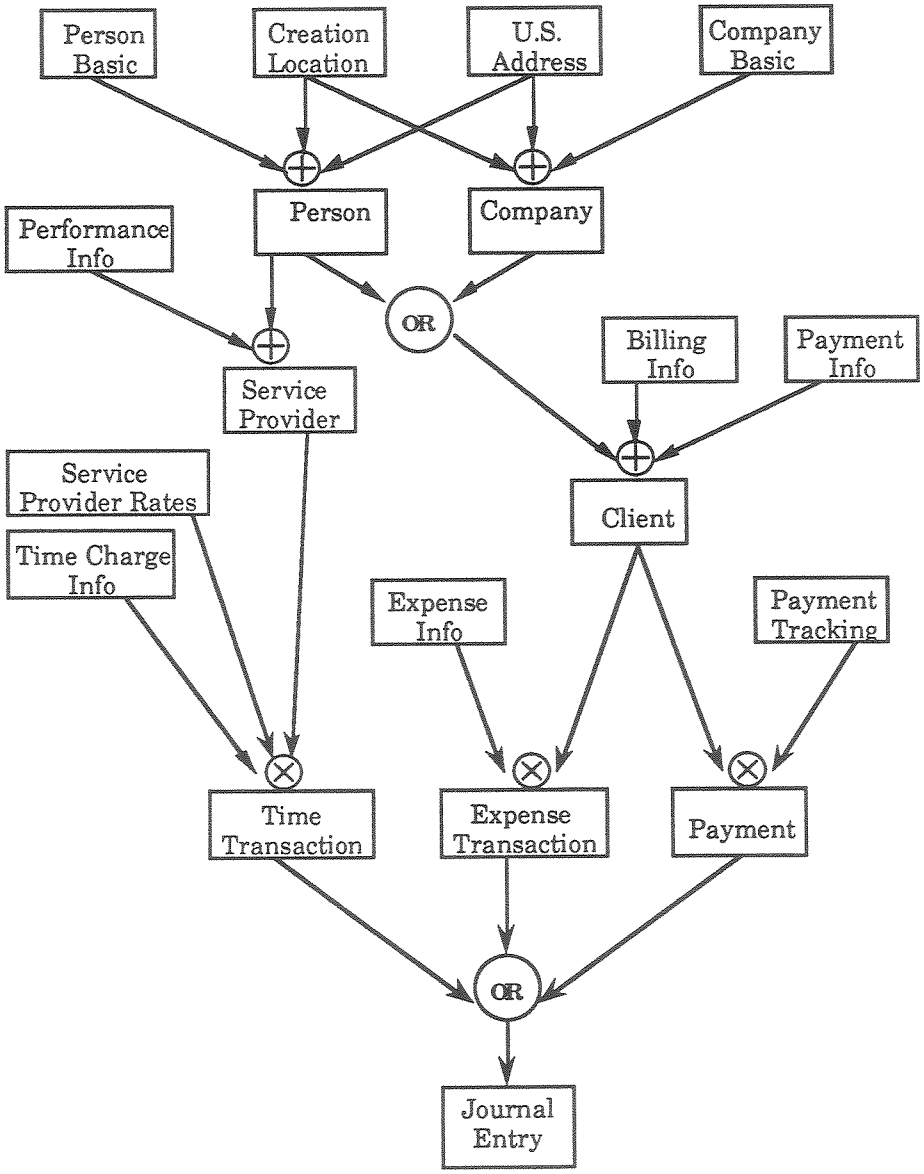
T1. A) Check out a copy of a book B) Return a copy of a book
T2. A) Add a copy of a book to B) Remove a a copy of a book from the library;
T3. Get the list of books by A) A particular author or B) In a particular subject area;
T4. Find out the list of books currently checked out be a particular borrower
T5. Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers. Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The data base must also satisfy the following constraints:

C1. All copies in the library must be available for checkout or be checked out.
C2. No copy of the book may be both available and checked out at the same time.
C3. A borrower may not have more than a predefined number of books checked out at one time

Appendix B – Accounts Receivable Example

The following listing of attributes and classes is a summary of the information for a domain model for a simple accounts receivables system for a professional time and billing application. Figure B.1 shows the class structure along with the required composition and associations.



B.1. Dates

B.1.1. Attributes

Julian_Date: Interval_scale Time in Days (min 1000) (max 800000)
Date_Current: Restricted Julian_Date (min 1990 year) (max 2050 year)
Date_Historical: Restricted Julian_Date (max 1990 year)

B.1.2. Class

```
Class = Date_mmddyy
    Month: Ratio_scale Time in Days (min 1) (max 12)
    Day: Ratio_scale Time in Days (min 1) (max 31)
    Year: Ratio_scale Time in Years (min 1900) (max 2100)
OPERATIONS (see c routines at end)
    cal_mjd (month, day, year, mjd)
    mjd_cal (mjd, month, day, year)
    mjd_dow (mjd, dow)
    mjd_dpm (mjd, ndays)
    mjd_year (mjd, year)
```

B.2. Money

GLDollar : Money in Dollars ((min -1000000) (max 1000000) (gran .01))
dollar_fee: Restricted GLDollar ((min 0) (gran 1))

B.3. Time

Hourly_1: Ratio_scale Time in Hours (min 0) (max 3000)
Hourly_0.25: Restricted Hourly_1 (min 0) (max 3000) (gran .25)
Hourly_0.1: Restricted Hourly_1 (min 0) (max 3000) (gran .1)

B.4. AR Legal System Attributes

Billing_rate: Ratio_Scale Money/Time in Dollars/Hour
(min 0) (max 1000) (gran .01)
Unit_Billing_rate : Restricted Billing_rate (gran 1)
Dollar_hour: Restricted Billing_rate (min 7) (max 500) (gran .05)
Client_Matter_Codes: Nominal_scale (contract, corporate, criminal,
estate_planning, family_practice, intellectual_property, labor, litigation,
personal_injury, real_estate tax, workmans_comp)

Type_Billing_time : Nominal_scale (billable, non_billable, vacation, education,
public_service);

Billing_level : Ordinal_scale(clerk_typist, secretary, paralegal, summer_clerk,
other_clerk, associate, sr_associate, jr_partner, partner, sr_partner)

Fee_Type: Nominal_Scale (flat_fee, flat_fee_w_retainer, hourly_rate_retainer,
retainer_w_max_hours, contingency, nonbillable,
flat_fee_w_max_dollars, retainer_w_max_dollars)

Service_code: Nominal_scale (telephone_consultation, personal_consultation,
court_time, title_search, audit, legal_research, on_site_research, deposition,
witness_interview, document_preparation, negotiation,
proposal_preparation, case_preparation, letter)

Expense_codes: Nominal_scale (travel, phone, copying, client_entertainment,
court_costs)

B.5. Defining a Person

Another case where composition can be applied is when using and reusing the information required to build classes for common business purposes. In the following example, creation_location is composed with either a basic_person class or a basic_company class to create a new class with more information. Figure 5.5 shows a creation_location class:

B.5.1. Person_basic

A person class can be defined in a variety of domain-specific ways. In this example, a basic person class *person_basic,* consists of a name and a social security number.

```
Person_basic = Class

NAMING ATTRIBUTES
    FName: Person_name;
    MName: Person_name;
    LName: Person_name;
    SSN: Nominal_scale:  Social_Sec_number;
```

B.5.2. Location

Chapter three introduced several types of location classes. Two are used in this definition of person.

```
US_Address = Class
    Address: US_street_address;
    City: US_City;
    State: US_State = Nominal_scale (AL, AK, AZ, AR, CA, CO, CT, DE, DC, FL,
            GA, HI, ID, IL, IN, IA, KS, KY, LA, ME, MD, MA, MI, MN, MS, MO, MT,
            NE, NJ, NV,NH, NM, NY, NC, ND, OH, OK, OR, PA, PR, RI, SC, SD,
            TN, TX, UT, VT, VA, WA, WV, WI, WY);
    Zip5 : Old_Zip_code;
    Zip4: New_Zip_code;
```

```
Creation_Location = Class
    Creation_date:  Gregorian_date;
    City_of_Creation:  US_City;
    State_of_creation:  US_State;
```

Creation_location is a class without a naming attribute that can be used in a variety of different situations. When it is composed with a basic_company class a company class is created as shown in Figure 5.7. The composition process is indicated as follows:

Company Class : = (⊕ company_basic creation_location)

```
Company = Class
NAMING ATTRIBUTES
    Name: Company_name;
    Federal_Tax_Identification_number;

NON-NAMING ATTRIBUTES
    Incorporation date: Julian_date;
    City_of_incorporation: Nominal_scale;
    State_of_incorporation: State_Type;
    Address: US_street_address;
    City: US_City;
    State: US_State
    Zip5 : Old_Zip_code;
    Zip4: New_Zip_code;
    Phone: phone_number;
```

Company_Class: = Company_Class : = (⊕ company_basic creation_location mailing_address)

The creation location can also be composed with a person to create a person class:

person_class: = (⊕ person_basic creation_location).

This operation is shown as being carried out in Figures 5.8 and 5.9.

B.5.3. Person = (⊕ Person_basic creation_location US_address)

For the purposes of this example, person is composed from three primitive classes. Obviously other classes could be added into the composition if they were determined to be necessary by the domain expert.

```
Person = Class

NAMING ATTRIBUTES
    FName: Person_name;
    MName: Person_name;
    LName: Person_name;
    SSN: Nominal_scale: Social_Sec_number;

NON-NAMING ATTRIBUTES
    date_of_birth: Juliian_Date
    City_of_birth: Nominal_scale;
    State_of_birth: State_Type;
    Address: US_street_address;
    City: US_City;
    State: US_State
    Zip5 : Old_Zip_code;
    Zip4: New_Zip_code;
    Phone: phone_number;
```

Person_Class: = (⊕ Person_basic creation_location)

B.6. Company

A Company is defined in a manner analogous to that of a person. For the purposes of this domain model, the difference between a company and a person is found primarily in their naming attributes.

```
Company_basic = Class

NAMING ATTRIBUTES
    Name: Company_name;
    Federal Tax_ID : Nominal_scale;
```

B.6.1. Company_Class : = (⊕ company_basic creation_location mailing_address)

```
Company = Class

NAMING ATTRIBUTES
    Name: Company_name;
    Federal_Tax_Identification_number;

NON-NAMING ATTRIBUTES
    Incorporation_date: Julian_date;
    City_of_incorporation: Nominal_scale;
    State_of_incorporation: State_Type;
    Address: US_street_address;
    City: US_City;
    State: US_State
    Zip5 : Old_Zip_code;
    Zip4: New_Zip_code;
    Phone: phone_number;
```

B.7. Service Provider

A Service Provider is a person who provides a service to a client. *Performance_info* is a class which contains the information necessary for a firm to track the financial performance of a service provider.

B.7.1. Performance_info

```
Class = Performance_info
NON-NAMING ATTRIBUTES
    unbilled_hours: Hourly_0.25,
    unbilled_amount: Fee,
    billed_hours: Hourly_0.25,
    billed_amount: Fee,
    non_billable_hours: Hourly_0.25,
    nonbillable_amount: Fee,
    hours_written_off: Hourly_0.25,
    amount_written_off: Fee,
    number_write_offs,
    contingent_hours: Hourly_0.25,
    contingent_amount: Fee,
    expended_hours: Hourly_0.25,
    expended_amount: Fee,
```

B.7.2. Service_Provider := (⊕ Person Performance_info)

```
Class = Service_Provider
NAMING ATTRIBUTES
    FName: Person_name;
    MName: Person_name;
    LName: Person_name;
    SSN: Nominal_scale: Social_Sec_number;
NON-NAMING ATTRIBUTES
    Billing_level : Ordinal_scale( clerk_typist, secretary, paralegal, summer_clerk,
    other_clerk, associate, sr_associate, jr_partner, partner, sr_partner)
    date_hired: date;
    Address: US_street_address;
    City: US_City;
    State: US_State
    Zip5 : Old_Zip_code;
    Zip4: New_Zip_code;
    Phone: phone_number;
    unbilled_hours: Hourly_0.25,
    unbilled_amount: Fee,
    billed_hours: Hourly_0.25,
    billed_amount: Fee,
    non_billable_hours: Hourly_0.25,
    nonbillable_amount: Fee,
    hours_written_off: Hourly_0.25,
    amount_written_off: Fee,
    number_write_offs,
    contingent_hours: Hourly_0.25,
    contingent_amount: Fee,
    expended_hours: Hourly_0.25,
    expended_amount: Fee,
```

B.8. Defining a Client

A client is a person or a company composed with billing and payment information.

B.8.1. Payment_info

```
Payment_Info = Class
    date_opened: date;
    last_statement_date: Julian_date;
    last_payment_date: Julian_date;
    last_payment_amount:: GLDollar;
    payments_ytd: GLDollar;
    payments_total_td: GLDollar;
DERIVED ATTRIBUTE
    days_overdue: currrent_date - last_payment_date
```

B.8.2. Billing_Info

```
Billing_Info = Class
    Work_description: text
    Budget_hours: Hourly_1.0 (max 200)
    Budget_amount (min 250) (max 100000)
    Fees_due: GLdollar;
    Expenses_due: GLDollar;
    Advances_due: GLDollar;
    Finance_charge_due: GLDollar;
    Fee_sales_tax_due: GLDollar;
    Expense_sales_tax_due,: GLDollar;
    billed_work_in_progress: GLDollar;
    billed_wip_tax: GLDollar;
    excess_payments: GLDollar;

DERIVED ATTRIBUTE
    Balance_Due:= Fees_due + Expenses_due + Advances_due +    Finance_charge_due
+ Fee_sales_tax_due + Expense_sales_tax_due, +          billed_work_in_progress +
billed_wip_tax - excess_payments
```

B.8.3. Client := (⊕ Payment_Info Billing_Info Company)

```
Client = Class (company)
NAMING ATTRIBUTES
    Name: Company_name;
    Federal_Tax_Identification_number;
NON-NAMING ATTRIBUTES
    Incorporation date: Julian_date;
    State_of_incorporation:  State_Type;
    date_opened: date;
    Address: US_street_address;
    City: US_City;
    State: US_State
    Zip4: New_Zip_code;
    Phone: phone_number;
    last_statement_date: Julian_date;
    last_payment_date: Julian_date;
    last_payment_amount:: GLDollar;
    payments_ytd: GLDollar;
    payments_total_td: GLDollar;
    Work_description: text
    Budget_hours: Hourly_1.0 (max 200)
    Budget_amount (min 250) (max 100000)
    Fees_due: GLdollar;
    Expenses_due: GLDollar;
    Advances_due: GLDollar;
    Finance_charge_due: GLDollar;
    Fee_sales_tax_due: GLDollar;
    Expense_sales_tax_due,: GLDollar;
    billed_work_in_progress: GLDollar;
    billed_wip_tax: GLDollar;
    excess_payments: GLDollar;
DERIVED ATTRIBUTE
    Days_overdue: currrent_date - last_payment_date
Balance_Due:= Fees_due + Expenses_due + Advances_due + Finance_charge_due +
        Fee_sales_tax_due + Expense_sales_tax_due, + billed_work_in_progress +
        billed_wip_tax - excess_payments
```

B.8.4. Client := (⊕ Payment_Info Billing_Info Person)

```
Client = Class (person)
NAMING ATTRIBUTES
    FName: Person_name;
    LName: Person_name;
    SSN: Nominal_scale: Social_Sec_number;
NON-NAMING ATTRIBUTES
    date_of_birth: Juliian_Date
    State_of_birth: State_Type;
    date_opened: date;
    Address: US_street_address;
    City: US_City;
    State: US_State
    Zip5 : Old_Zip_code;
    Zip4: New_Zip_code;
    Phone: phone_number;
    last_statement_date: Julian_date;
    last_payment_date: Julian_date;
    last_payment_amount:: GLDollar;
    payments_ytd: GLDollar;
    payments_total_td: GLDollar;
    Work_description: text
    Budget_hours: Hourly_1.0 (max 200)
    Budget_amount (min 250) (max 100000)
    Fees_due: GLdollar;
    Expenses_due: GLDollar;
    Advances_due: GLDollar;
    Finance_charge_due: GLDollar;
    Fee_sales_tax_due: GLDollar;
    Expense_sales_tax_due,: GLDollar;
    billed_work_in_progress: GLDollar;
    billed_wip_tax: GLDollar;
    excess_payments: GLDollar;
DERIVED ATTRIBUTE
    Days_overdue: currrent_date - last_payment_date
    Balance_Due:= Fees_due + Expenses_due + Advances_due +    Finance_charge_due
+ Fee_sales_tax_due + Expense_sales_tax_due, +          billed_work_in_progress +
billed_wip_tax - excess_payments
```

B.9. Time Transaction

A time_transaction class requires a Service_provider_rates class and a Time_charge_Info class.

B.9.1. Service_provider_rates

```
Class = Service_provider_rates

 NAMING ATTRIBUTES
    Billing_level : Ordinal_scale (clerk_typist, secretary, paralegal, summer_clerk,
    other_clerk, associate, sr_associate, jr_partner, partner, sr_partner)
 NON-NAMING ATTRIBUTES
    Billing_rate_regular: Dollars_per_hour ((min 35) (max 250) (gran 1)) ;
    Billing_rate_premium: Dollars_per_hour ((min 35) (max 250) (gran 1));
```

B.9.2. Time Charge Information

```
Class = Time_charge_Info
NAMING ATTRIBUTES
    Time_Transaction_Number: sys_num;

NON-NAMING ATTRIBUTES
     control_time: sys_time;
    control_date: sys_date
    Service_code: Nominal_scale ( telephone_consultation, personal_consultation,
    court_time, title_search, audit, legal_research, on_site_research, deposition,
    witness_interview, document_preparation, negotiation, proposal_preparation,
    case_preparation, letter)
    charge_level Nominal_scale (fixed, regular, premium);
    minimum: boolean;
    min_hours : work_hours (min 0) (max 25);
    description: text;

 AXIOMS
    IF minimum
            THEN mustenter(min_hours)
            ELSE not_applicable(min_hours);
```

B.9.3. Time_transaction:=
(⊗ Service_provider Service_provider_rates Time_charge_Info Client)

```
Class = Time_transaction

EXTERNAL CLASSES
    CLASS Service_provider (Name Employee_name) (CARD M);
    CLASS  Client (Name Client_name)(CARD 1);
    CLASS  Service_provider_rates(Name Billing_level ) (CARD 1);
    CLASS  Time_charge_info (Name Time_Transaction_Number)  (CARD N);

NAMING ATTRIBUTES
    Time_Transaction_Number: sys_num;

NON-NAMING ATTRIBUTES
    Hours_billed: hourly_.25,
    Billing_type: Type_billing_time
    Billing_charge_unit_hour: f(billing_type, Billing_level),
    Billing_amt: Dollar_amount,
    Description: Text,
    elapsed_time: hourly_0.25;
    description: text;
```

B.10. Expenses

Expenses are charges such as copying costs that are recorded independently of time transactions.

B.10.1. Expense_Info

```
Class = Expense_info

NAMING ATTRIBUTES
    Expense_Transaction_Number: sys_num
NON-NAMING ATTRIBUTES
    Expense_codes: Nominal_scale ( travel, phone, copying, client_entertainment,
    court_costs)
    control_date: sys_date
    calculate_per_unit, use_minimum: mutex_boolean;
    sales_tax: boolean;
    fee_per_unit: dollar_fee (min .25) (max 250) (gran .25);
    min_fee: dollar_fee (min 100) (max 5,000)(gran .1) ;
    billable : boolean;
    printable: boolean;
    gl_accnt_number:
    description: text;

AXIOMS
IF calculate_per_unit:
    THEN BEGIN
            mustenter(fee_per_unit:);
            END
    ELSE not_applicable(fee_per_unit:);
IF use_minimum
    THEN BEGIN
            mustenter(min_fee);
    ELSE not_applicable(min_fee);
```

B.10.2. Expense_Transaction := (⊗ Expense_info Client)

```
Class = Expense_Transaction

EXTERNAL CLASSES
    CLASS  Expense_Info (Name Expense_Transaction_Number) (CARD M);
    CLASS  Client (Name Client_name) (CARD 1);

NAMING ATTRIBUTES
    Expense_Transaction_Number: sys_num

NON-NAMING ATTRIBUTES
    Expense_codes: Nominal_scale ( travel, phone, copying, client_entertainment,
    court_costs)
    control_date: sys_date
    calculate_per_unit, use_minimum: mutex_boolean;
    sales_tax: boolean;
    fee_per_unit: dollar_fee (min .25) (max 250) (gran .25);
    min_fee: dollar_fee (min 100) (max 5,000)(gran .1) ;
    billable : boolean;
    printable: boolean;
    gl_accnt_number:
    description: text;
```

B.11. Payments

Payments are made by the client to offset charges created by time and expense transactions.

B.11.1. Payment_Tracking

```
Class = Payment_Tracking

NAMING ATTRIBUTES
    Payment_Number: system_num

NON-NAMING ATTRIBUTES
    control_date: sys_date
    Type_of_payment: Nominal_scale (check, money_order, bank_draft);
    amt_of_payment: GLDollar;
    gl_accnt_number:
    description: text;
```

B.11.2. Payment := (⊗ Payment_Tracking Client)

```
Class = Payment

EXTERNAL CLASSES
    CLASS  Payment_Tracking (Name Payment_Number) (CARD M);
    CLASS  Client (Name Client_name) (CARD 1);

NAMING ATTRIBUTES
    Payment_Number: system_num

NON-NAMING ATTRIBUTES
    control_date: sys_date
    Type_of_payment: Nominal_scale (check, money_order, bank_draft);
    amt_of_payment: GLDollar;
    gl_accnt_number:
    description: text;
```

B.11.3. Journal Entry

Time transactions, expense transactions, and payments can all be viewed as journal entries. Journal_entry := (OR Time_transaction Expense_transaction Payment)

```
Class = Journal_entry (Time_transaction)

EXTERNAL CLASSES
    CLASS  Time_transaction(Name Transaction_Number) (CARD 1);

NAMING ATTRIBUTES
    Journal_control: system_num

NON-NAMING ATTRIBUTES
    control_date: sys_date;
    amt: GLDollar;
    gl_accnt_number_debit: gl_acct_number;
    gl_accnt_number_credit: gl_acct_number;
    description: text;

Class = Journal_entry (Payment )

EXTERNAL CLASSES
    CLASS  Payment (Name Payment_Number) (CARD 1);

NAMING ATTRIBUTES
    Journal_control: system_num

NON-NAMING ATTRIBUTES
    control_date: sys_date;
    amt: GLDollar;
    gl_accnt_number_debit: gl_acct_number;
    gl_accnt_number_credit: gl_acct_number;
    description: text;

Class = Journal_entry (Expense_info)

EXTERNAL CLASSES
    CLASS  Expense_info(Name Expense_Transaction_Number) (CARD 1);

NAMING ATTRIBUTES
    Journal_control: system_num

NON-NAMING ATTRIBUTES
    control_date: sys_date;
    amt: GLDollar;
    gl_accnt_number_debit: gl_acct_number;
    gl_accnt_number_credit: gl_acct_number;
    description: text;
```

B.12. Date Routines in C

The following C routines do calculations in modified Julian dates. To get actual Julian date:

```
double jd = mjd + 2415020L;
```

```
#include <stdio.h>
#include <math.h>
#include "astro.h"

/* given a date in months, month, days, day, years, year,
 * return the modified Julian date (number of days elapsed since 1900 jan 0.5),
 * *mjd.
 */
cal_mjd (month, day, year, mjd)
int month, year;
double day;
double *mjd;
{
        int b, d, m, y;
        long c;

        m = month;
        y = (year < 0) ? year + 1 : year;
        if (month < 3) {
           m += 12;
           y -= 1;
        }

        if (year < 1582 || year == 1582 && (month < 10 || month == 10 && day < 15))
           b = 0;
        else {
           int a;
           a = y/100;
           b = 2 - a + a/4;
        }

        if (y < 0)
           c = (long)((365.25*y) - 0.75) - 694025L;
        else
           c = (long)(365.25*y) - 694025L;

        d = 30.6001*(m+1);

        *mjd = b + c + d + day - 0.5;
}

/* given the modified Julian date (number of days elapsed since 1900 jan 0.5,),
 * mjd, return the calendar date in months, *month, days, *day, and years, *year.
 */
mjd_cal (mjd, month, day, year)
double mjd;
```

```
int *month, *year;
double *day;
{
        double d, f;
        double i, a, b, ce, g;

        d = mjd + 0.5;
        i = floor(d);
        f = d-i;
        if (f == 1) {
           f = 0;
           i += 1;
        }

        if (i > -115860.0) {
           a = floor((i/36524.25)+.9983573)+14;
           i += 1 + a - floor(a/4.0);
        }

        b = floor((i/365.25)+.802601);
        ce = i - floor((365.25*b)+.750001)+416;
        g = floor(ce/30.6001);
        *month = g - 1;
        *day = ce - floor(30.6001*g)+f;
        *year = b + 1899;

        if (g > 13.5)
           *month = g - 13;
        if (*month < 2.5)
           *year = b + 1900;
        if (*year < 1)
           *year -= 1;
}

/* given an mjd, set *dow to 0..6 according to which day of the week it falls
 * on (0=sunday) or set it to -1 if can't figure it out.
 */
mjd_dow (mjd, dow)
double mjd;
int *dow;
{
        /* cal_mjd() uses Gregorian dates on or after Oct 15, 1582.
         * (Pope Gregory XIII dropped 10 days, Oct 5..14, and improved the leap-
         * year algorithm). however, Great Britian and the colonies did not
         * adopt it until Sept 14, 1752 (they dropped 11 days, Sept 3-13,
         * due to additional accumulated error). leap years before 1752 thus
         * can not easily be accounted for from the cal_mjd() number...
         */
        if (mjd < -53798.5) {
           /* pre sept 14, 1752 too hard to correct */
           *dow = -1;
           return;
        }
        *dow = ((int)floor(mjd-.5) + 1) % 7; /* 1/1/1900 (mjd 0.5) is a Monday*/
```

```
        if (*dow < 0)
            *dow += 7;
}

/* given a mjd, return the the number of days in the month.  */
mjd_dpm (mjd, ndays)
double mjd;
int *ndays;
{
        static short dpm[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
        int m, y;
        double d;

        mjd_cal (mjd, &m, &d, &y);
        *ndays = (m==2 && ((y%4==0 && y%100!=0)||y%400==0)) ? 29 : dpm[m-];
}
/* given a mjd, return the year as a double. */
mjd_year (mjd, year)
double mjd;
double *year;
{
        int m, y;
        double d;
        double e0, e1;  /* mjd of start of this year, start of next year */

        mjd_cal (mjd, &m, &d, &y);
        cal_mjd (1, 1.0, y, &e0);
        cal_mjd (12, 31.0, y, &e1); e1 += 1.0;
        *year = y + (mjd - e0)/(e1 - e0);
}
```

Appendix C – Attribute Grammar (Partial)

The following productions are intended to resolve ambiguities in attribute arithmetic. More complete explanations are given in Chapter 3.

(Plus A_1 + A_2) -> A_3

```
1.  Procedure Add_attribute(A1,A2:Attribute;VAR A3:Attribute)
2.  Begin
3.  IF (A1.Scale = RATIO AND A2.Scale = RATIO)
4.    AND (A1.quantity = A2.quantity) AND COMPARABLE (A1.T,
            A2.T)
5.  Then Begin
6.      If  (A1.unit = A2.unit)
7.      Then
8.              If  (A1.gran = A2.gran)
9.              Then Begin
10.                             A3.Scale := RATIO;
11.                             A3.quantity := A2.quantity;
12.                             A3.unit := A2.unit;
13.                             A3.gran = A2.gran;
14.                             A3.T  := A1.T;
15.                             A3.min := A1.min + A2.min;
16.                             A3.max := A1.max + A2.max;
17.                             build_scale(A3.min,A3.max,
                                                A3.gran,A3.V);
18.                             A3.EV := A1.EV  + A2.EV;
19.                             A3.SD2 := A1.SD2  + A2.SD2
                                {assumes independence}
20.                             End
21.                     Else Begin
22.                             coerce_gran(A1,A2, Aprime1, Aprime2);
23.                             Add_attribute(Aprime1, Aprime2, A3);
24.                             End
25.             Else Begin
26.                     coerce_unit(A1, A2, Aprime1, Aprime2, error);
27.                     add_attribute(Aprime1, Aprime2, A3);
28.                     End
29.     Else ERROR( "Mismatched Scales")
30.     End; {add_attribute}

31.     Procedure coerce_unit(A1,A2:Attribute; VAR Aprime1,
                                                Aprime2:Attribute,
             VAR error:boolean);
32.     Begin
33.             If bigger_unit(A1.unit, A2.unit)
34.             then begin
35.                     new_att_new_unit( A2, A1.unit, A2prime)
36.                     A1prime := A1;
37.                     end
38.             else begin
39.                     new_att_new_unit( A1, A2.unit, A1prime);
40.                     A2prime := A2;
41.                     end
```

```
42.   End; {coerce unit)

43.   Procedure new_att_new_unit(oldatt:Attribute; newunit:TyUnit;
            newatt:Attribute, VAR error:boolean);
44.   Begin
45.         newatt.T  := oldatt.T;
46.         newatt.Scale := oldatt.Scale;
47.         newatt.quantity := old_att.quantity;
48.         newatt.unit := old_att.newunit;
49.         unitfactor( oldatt.unit, newunit, factor);
50.         newatt.gran = old_att.gran/factor;
51.         newatt.min := oldatt.min * newatt.gran;
52.         newatt.max := oldatt.max * newatt.gran;
53.         build_scale(newatt.min, newatt.max, newatt.gran,
                                                newatt.V);
54.         newatt.EV := old_att.EV * newatt.gran;
55.         newatt.SD2 := old_att.SD2 * newatt.gran**2;
56.   End; {new_att_new_unit}

57.   Procedure coerce_gran(A1,A2:Attribute; VAR Aprime1,
                                    Aprime2:Attribute);
58.   Begin
59.         If bigger_gran(A1.gran, A2.gran)
60.         then begin
61.               new_att_new_gran( A2, A1.gran, A2prime)
62.               A1prime := A1;
63.               end
64.         else begin
65.               new_att_new_gran( A1, A2.gran, A1prime)
66.               A2prime := A2; end
67.   End; {coerce gran}

68.   Procedure new_att_new_gran(oldatt:Attribute;
                  newgran:TyUnit; newatt:Attribute);
69.   Begin
70.         newatt.T  := oldatt.T;
71.         newatt.Scale := oldatt.Scale;
72.         newatt.quantity := old_att.quantity;
73.         newatt.unit := old_att.unit;
74.         newatt.gran = newgran;
75.         factor := newgran/oldatt.gran;
76.         newatt.min := oldatt.min * factor;
79.         newatt.max := oldatt.max * factor;
80.         build_scale(newatt.min, newatt.max, newatt.gran,
                                                newatt.V);
81.         newatt.EV := old_att.EV * factor;
82.         newatt.SD2 := old_att.SD2 * factor**2;
83.   End; {new_att_new_gran}

84.   Procedure unit_factor(oldunit, newunit, VAR unitfactor,
                                    VAR error:boolean);
85.   Begin
86.         Query(oldunit, newunit, unitfactor);
87.   End; {unit_factor)
```

```
88.   Procedure build_scale(A3.min, A3.max, A3.gran; VAR A3.V);
89.   Begin
90.         value = A3.min
91.         A3.V = Null
92.         Repeat
93.               A3.V := Union(A3.V, value)
94.               value := value + A3.gran
95.         Until value > A3.max
96.   End; {build_scale)
```

(Plus A_1 C) -> A_3

```
1.  Procedure Add_attribute_plus_C(A1,C:Attribute;VAR A3:Attribute)
2.  Begin
3.  IF ((A1.Scale = RATIO AND C.Scale = RATIO)  OR
        (A1.Scale = INTERVAL AND C.Scale = INTERVAL))
4.      AND (A1.quantity = C.quantity)
5.  Then Begin
6.      If  (A1.unit = C.unit)
7.      Then
8.            If  (A1.gran = C.gran)
9.            Then Begin
11.                     A3.quantity := A1.quantity;
12.                     A3.unit := A1.unit;
13.                     A3.gran = A1.gran;
14.                     A3.T  := A1.T;
15.                     A3.min := A1.min + C;
16.                     A3.max := A1.max + C;
17.                     build_scale(A3.min,A3.max,
                                          A3.gran,A3.V);
18.                     A3.EV := A1.EV  + C;
19.                     A3.SD2 := A1.SD2;
                                    {assumes independence}
20.                     End
21.               Else Begin
22.                     coerce_gran(A1,C, Aprime1, Aprime2);
23.                     Add_attribute(Aprime1, Aprime2, A3);
24.                     End
25.           Else Begin
26.                 coerce_unit(A1, C, Aprime1, Aprime2, error);
27.                 add_attribute(Aprime1, Aprime2, A3);
28.                 End
29.     Else ERROR( "Mismatched Scales")
30.     End; {add_attribute_plus_C}
```

(Times A_1 C) -> A_3

```
1.  Procedure Times_Attribute_C(A1,C:Attribute;VAR A3:Attribute)
2.  Begin
3.  IF ((A1.Scale = RATIO AND C.Scale = RATIO)  OR
        (A1.Scale = INTERVAL AND C.Scale = INTERVAL))
4.      AND (A1.quantity = C.quantity)
5.  Then Begin
```

```
6.      If  (A1.unit = C.unit)
7.      Then
8.              If  (A1.gran = C.gran)
9.              Then Begin
11.                             A3.quantity := A1.quantity;
12.                             A3.unit := A1.unit;
13.                             A3.gran = A1.gran;
14.                             A3.T  := A1.T;
15.                             A3.min := A1.min * C;
16.                             A3.max := A1.max * C;
17.                             build_scale(A3.min,A3.max,
                                                A3.gran,A3.V);
18.                             A3.EV := A1.EV * C;
19.                             A3.SD2 := A1.SD2 * C**2;
20.                             End
21.                     Else Begin
22.                             coerce_gran(A1,C, Aprime1, Aprime2);
23.                             Add_attribute(Aprime1, Aprime2, A3);
24.                             End
25.             Else Begin
26.                     coerce_unit(A1, C, Aprime1, Aprime2, error);
27.                     add_attribute(Aprime1, Aprime2, A3);
28.                     End
29.     Else ERROR( "Mismatched Scales")
30.     End; {add_attribute_plus_C}
```

(Sub A_1 A_2) -> A_3

```
1.  Procedure subtract_attribute(A1, A2, A3)
2.  Begin
3.  IF(A1.quantity = A2.quantity) AND COMPARABLE (A1.T, A2.T)
4   AND ((A1.Scale = RATIO AND A2.Scale = RATIO)  OR
            (A1.Scale  = INTERVAL AND A2.Scale = INTERVAL))
5.  Then Begin
6.      If  (A1.unit = A2.unit)
7.      Then
8.              If  (A1.gran = A2.gran)
9.              Then Begin
10.                     A3.Scale := RATIO;
11.                     A3.quantity := A2.quantity;
12.                     A3.unit := A2.unit;
13.                     A3.gran = A2.gran;
14.                     A3.min := max(0, A1.min - A2.min);
15.                     A3.max := A1.max - A2.max;
16.                     build_scale(A3.min, A3.max,A3.gran, A3.V);
17.                     A3.T  := A1.T
18.                     A3.EV := A1.EV - A2.EV;
19.                     A3.SD2 := A1.SD2 + A2.SD2; {assume ind}
20.                     End
21.             Else Begin
22.                     coerce_gran(A1, A2, Aprime1, Aprime2);
23.                     Add_attribute(Aprime1, Aprime2, A3);
24.                     End
25.             Else Begin
```

```
26.                 coerce_unit(A1, A2, Aprime1, Aprime2, error);
27.                 add_attribute(Aprime1, Aprime2, A3);
28.                 End
29.     Else ERROR( "Mismatched Scales")
30.     End; {subtract_attribute}
```

The following productions are intended to give the flavor of the semantics of $\mathcal{PP}$(A).introduced in chapter 2.

```
Pop_parm ⇒
        minimum, maximum, mean, variance, median, mode

pdf ⇒       ε
        | uniform
        | normal
        | poisson
        | exponential
        | gamma
        | chi_square

 minimum ⇒ (min   num)
 maximum ⇒ (max   num)

            min < max;
            if A1.Scale = RATIO then min = 0;

mean ⇒ (MEAN μ)        |
            min <= μ <= max

variance ⇒ (VARIANCE       σ2)    |
            IF σ2 = 0
                Then
                    min_num=max_num= μ = M = Mo
                    Prob( |A1.value| = μ ) = 1
                Else
                    ∀ x Prob(|A1.value - μ| >= x σ ) <= x-2
                    {Chebyshev's inequality distribution unknown or not normal}
                    Prob( |A1.value| > μ + 2σ ) < 0.75
                    Prob( |A1.value| > μ + 3σ ) < 0.89
median ⇒ (MEDIAN      M)     |
            min < M < max
mode   ⇒ (MODE   Mo)    |
quartiles   ⇒ (QUARTILES q1 q2 q3)
            q2 = M
deciles   ⇒ (DECILES d1 d2 d3 d4 d5 d6 d7 d8 d9 d10)
            d5 = M
```

uniform $\Rightarrow$ (UNIFORM C)

$\mu = C$

$\sigma^2 = 0$

normal $\Rightarrow$ (NORMAL μ σ^2) |

$\mu = M = Mo$

Prob($|A_1.value| > \mu + 1.96\sigma$) < 0.05

Prob($|A_1.value| > \mu + 2.5\sigma$) < 0.01

poisson $\Rightarrow$ (POISSON α) |

$\mu = \alpha$

$\sigma^2 = \alpha$

exponential $\Rightarrow$ (EXPONENTIAL α) |

$f(x) = \alpha e^{-\alpha x}$

$\mu = 1/\alpha$

$\sigma^2 = 1/\alpha^2$

gamma $\Rightarrow$ (GAMMA r α) |

$r > 0$

$\alpha > 0$

If r= 1 then distribution = EXPONENTIAL

If r = n/2 AND α = .5 then distribution = CHI_SQUARE

chi_square $\Rightarrow$ (CHI_SQUARE n) |

$\mu = n$

$\sigma^2 = 2 * n$

Appendix D – Definitions

An attribute, A, consists of :
- a unique name $\mathcal{N}(A)$
- a measurement scale $\mathcal{M}(A)$ and (when appropriate) unit & granularity
- a set of values $\mathcal{V}(A)$
- a set of population parameters $\mathcal{PP}(A)$
- an Initialization procedure $\mathcal{I}(A)$
- a probability of change $\mathcal{PC}(A)$
- a state transition relation $\mathcal{R}(A)$, $\mathcal{V}(A) \times \mathcal{V}(A) \supseteq \mathcal{R}(A)$

Definition 2.1 — Attribute

A nominal measurement scale m is a set of categories $\mathcal{C}(m) = \{C_1, \ldots C_n\}$.

Definition 2.2 – Nominal Scale

An ordinal scale is a nominal scale in which a total ordering exists among the categories C_i.

Definition 2.3 – Ordinal Scale

A *quantity* is defined in terms of:
- a unit defined in domain-specific operational terms
- a measurement granularity, that is the highest degree of precision to which a unit is normally expressed within a domain.

Definition 2.4 — Quantity

An interval scale is a nominal scale that has an associated quantity and assigns a unique multiple (called the magnitude) of the measurement granularity of the unit of the quantity to each category.

Definition 2.5 — Interval Scale

A ratio scale is an interval scale that has a non-arbitrary zero and allows only non-negative magnitudes

Definition 2.6 — Ratio Scale

For a set of categories $\mathcal{C}(m) = \{C_1, \ldots C_n\}$ in a nominal scale $\mathcal{N}(m)$,
the default category C_i is the $\max(\text{Proportion}(C_i), \ldots, \text{Proportion}(C_n))$.

where $\text{Proportion}(C_i) = |C_i|/|\{C_1, \ldots C_n\}|$.

Definition 2.7 — Rule to Determine Default Category for a Nominal Scale

A is an immutable attribute if $\mathcal{PC}(A) = 0$ $\quad \mathcal{PC}(A) = 0 \dashrightarrow \mathcal{R}(A) = \phi$

Definition 2.8 — Immutable Attributes

Identity $\quad (x, x) \wedge x \in V(A)$

Definition 2.9 – Identity Relation

Single Step Increment	$(x, y) \dashrightarrow y \neq succ(x)$
Multi Step Increment	$(x, y) \dashrightarrow y \neq succ(x) \wedge x << y$
Single Step Decrement	$(x, y) \dashrightarrow y \neq pred(x)$
Multi Step Decrement	$(x, y) \dashrightarrow y \neq pred(x) \wedge y >> x$

Definition 2.10 — Additional Ordinal Scale Transition Pairs

Unit addition	$(x, x + 1) \wedge (x \in V) \wedge (x + 1 \in V)$
Arbitrary increment	$(x, x + i) \wedge (x \in V) \wedge (x + i \in V) \wedge (i > 0)$
Unit subtraction	$(x, x - 1) \wedge (x \in V) \wedge (x - 1 \in V)$
Arbitrary decrement	$(x, x - i) \wedge (i \in V) \wedge (x - i \in V) \wedge (i > 0)$

Definition 2.11 —Relation Subsets of Interest in an Interval Scale.

Multiplicative increase	$(x, x * i) \wedge (x \in V) \wedge (x * i) \in V) \wedge (i > 0)$
Multiplicative decrease	$(x, x * i) \wedge (x \in V) \wedge (x * i) \in V) \wedge (0 < i < 1)$

Definition 2.12 — Additional Subsets of Interest in a Ratio Scale

A class C is:

- a set of attributes $\mathcal{A}(C)$ and their subsets,
  - Primitive Attributes $\mathcal{P}(C)$,
  - Derived Attributes $\mathcal{D}(C)$,
  - Naming Attributes $\mathcal{N}(C)$,
  - Referential Attributes $\mathcal{R}(C)$,
- a set of functions $\mathcal{F}(C)$ that are used to create $\mathcal{D}(C)$,
  - These functions are expressed in terms of attributes.
- a set of Operations as Follows:
  - Object Instantiation $\mathcal{I}(C)$,
  - Object Removal $\mathcal{R}(C)$,
  - Other Operations $\mathcal{O}(C)$,
- a statistical summary function $\mathcal{S}(C)$,
- a set of axioms or integrity constraints $\mathcal{X}(C)$

Definition 3.1 – Class

Given a class C with attributes $A_0, A_1, \ldots A_n$, an object O of Class C is defined to be a family of vectors of the form $[a_0, a_1, \ldots a_n]$ where $a_j \in \mathcal{V}(A_j)$and where the vectors have a common value (the name of O) on the naming attributes of C.

Definition 3.2 – Object

The set of attributes $\mathcal{A}(C)$, can be viewed as two disjoint subsets: The fundamental, or primitive, attributes $\mathcal{P}(C)$ and the derived attributes $\mathcal{D}(C)$;

$$\mathcal{A}(C) = \mathcal{P}(C) \cup \mathcal{D}(C) \text{ and } \mathcal{P}(C) \cap \mathcal{D}(C) = \phi.$$

The $\mathcal{D}(C)$ require the introduction of a set functions $\mathcal{F}(C)) = \{f_i\}$, such that if $\mathcal{P}(C) = \{P_0, P_1, \ldots P_n\}$, $\mathcal{D}(C) = \{D_0, D_1, \ldots D_m\}$, and S is the special class of system attributes (such as current date), and $\mathcal{DOM}(S)$ is the cross product of the value sets of the primitive attributes of S, then there is a function $f_i \in \mathcal{F}(C)$ such that f_i is an *onto* function and $f_i : \mathcal{V}(P_0)$ X . . . X $\mathcal{V}(P_n)$ X $\mathcal{DOM}(S) \rightarrow \mathcal{V}(D_i)$.

Definition 3.3 – Class (Primitive and Derived Attributes)

A subset, $\mathcal{N}(C)$ of the $\mathcal{A}(C)$ are the naming attributes. $A \in \mathcal{N}(C) \rightarrow \text{null} \notin \mathcal{V}(\mathcal{A})$

Definition 3.4 – Naming Attributes

A subset, $\mathcal{R}(C)$ of the $\mathcal{A}(C)$ are the referential attributes, or $\mathcal{R}(C)$.

$$(\mathcal{N}(C) \supseteq \mathcal{R}(C)) \vee (\mathcal{R}(C) \cap \mathcal{N}(C) = \phi)$$

Definition 3.5 – Referential Attributes

Classes include:

An instantiation procedure I that uniquely associates any object of class C with some tuple of $\mathcal{V}(N_0)$ X . . . X $\mathcal{V}(N_n)$ $(N_i \in \mathcal{N}(C))$.

A set of deletion or removal procedures , $\mathcal{R}(C)$, that remove the objects created by the instantiation procedure I.

A set of Operations, $O(C)$, that access or update sets of attributes

Definition 3.6 – Operations

Classes include a statistical function $\mathcal{S}(C)$ that computes summary statistics for the set of objects instantiated for C, and a set of functions $\mathcal{SA}(C, A_i)$ that compute summary statistics for each attribute of the set of objects.

Definition 3.7 – Class Statistical Functions

Classes include a set of axioms or integrity constraints $\mathcal{X}(C)$:

- interclass attribute axioms
  - intraobject attribute axioms
  - interobject attribute axioms
- interclass attribute axioms

Definition 3.8 – Class Axioms

C' is a subclass of C if:

1. There is an attribute, A, with $\mathcal{V}(A) \supset \mathcal{V}'(A)$
2. $P(C) \supseteq P(C')$ That is, the attributes of C' consist of at most the primitive attributes of C and additional attributes that can be derived from those primitives.

C is a superclass of C', if C' is a subclass of C

Definition 4.1 – Subclass and Superclass

An attribute B is a restriction of an attribute A (or A is an extension of B) if :

$\mathcal{V}(A) \supset \mathcal{V}(B)$

$\mathcal{R}(A) \supset \mathcal{R}(B)$

Definition 4.2 – Attribute Restriction

C_M is a class that is the composition of the classes $C_1\ C_2 \ldots C_n$. The algorithm, (in which the subscript ✍ refers to a function completed by the domain expert), is as follows:
$C_M := (\oplus\ C_1\ C_2 \ldots C_n)$

1. $\mathcal{A}(C_M) := \mathcal{A}(C_1) \cup \mathcal{A}(C_2) \cup \ldots \mathcal{A}(C_n))$, where $\cup$ is the disjoint union

2. $\forall x\ \exists y\ ((\mathcal{N}(\mathcal{A}(C_x)) \cap \mathcal{N}(\mathcal{A}(C_y)) \neq \varnothing) \rightarrow \text{conflict\_resolution}_{✍}(\mathcal{A}(C_x), \mathcal{A}(C_y))$

3. $\mathcal{A}(C_M) := \mathcal{A}(C_M)$ - irrelevant _attributes$_{✍}(C_M)$

4. $\mathcal{A}(C_M) := \mathcal{A}(C_M) \cup$ selected_new_attributes$_{✍}$(attribute_library)

Definition 5.1— Composition Process Algorithm

C_M is a class that is the association of the classes $C_1\ C_2 \ldots C_n$. The algorithm is as follows:
$C_M := (\otimes\ C_1\ C_2 \ldots C_n)$.

1. $\mathcal{A}(C_M) := \mathcal{N}(C_1) \cup \mathcal{N}(C_2) \cup \ldots \mathcal{N}(C_n))$
2. $\mathcal{A}(C_M) := \mathcal{A}(C_M) \cup$ selected_new_attributes$_{✍}$(attribute_library)

Definition 6.1— Association

Bibliography

[Adelson 85] Adelson, B. and Soloway, E. The role of domain experience in software design, *IEEE Transactions on Software Engineering*. Vol. 11, No. 11 (November 1985), 1351-1359.

[Agresti 86a] Agresti, W. "What are the new paradigms?" in *New Paradigms for Software Development* (eds. Agresti, W.W.), IEEE Computer Society, Washington, DC, 6-10.

[Agresti 86b] Agresti, W. "Framework for a flexible development process" in *New Paradigms for Software Development* (eds. Agresti, W.W.), IEEE Computer Society, Washington, DC (1986), 11-14.

[Alexander 64] Alexander, C. *Notes on the Synthesis of Form*, Harvard University Press, Cambridge, MA (1964).

[Almes 83] Almes, G., Borning, A. and Messinger, E. "Implementing a smalltalk-80 system on the intel 432: A feasibility study" in *Smalltalk-80: Bits of History, Words of Advice* (eds. Krasner, G.), Addison-Wesley, Reading, MA (1983), 299-322.

[Anderson 89] Anderson, J.S. and Fickas, S. A proposed perspective shift: Viewing specification design as a planning problem, Proceedings of the Fifth International Workshop on Software Specification and Design, Pittsburgh, PA (May 19-20, 1989), 177-184.

[Apple 85] Apple. Apple Computer, Inc. Addison-Wesley Publishing Company, Inc, Reading, Massachusetts (1985).

[Arango 89] Arango, G. "Domain analysis: From art form to engineering discipline," Proceedings of the Fifth International Workshop on Software Specification and Design, Pittsburgh, PA (May 19-20, 1989), 152-159.

[Bailin 89] Bailin, S.C. An object-oriented requirements specification method, *Communications of the ACM*. Vol. 32, No. 5 (May 1989), 608-623.

[Balzer 76] Balzer, R., Goldman, N. and Wile, D. On the transformational implementation approach to programming, Proceedings, Second International Conference on Software Engineering, New York: Computer Society Press (1976), 337-344.

[Balzer 78] Balzer, R. and Goldman, N. Informality in program specifications, *IEEE Transactions on Software Engineering*. Vol. 4, No. 2 (March 1978), 94-102.

[Balzer 79] Balzer, R. and Goldman, N. Principles of good software specification and their implications for specification languages, Specification of Reliable Software Conference (1979), 58-67.

[Balzer 81] Balzer, R. Transformational implementation: An example, *IEEE Transactions on Software Engineering*. Vol. SE-7, No. 1 (January 1981), 3-14.

[Balzer 83] Balzer, R., Cheatham, T.E., Jr. and Green, C. Software technology in the 1990's: Using a new paradigm, *IEEE Computer* (November 1983).

[Balzer 85] Balzer, R. A 15 year perspective on automatic programming, *IEEE Transactions on Software Engineering*. Vol. SE-11, No. 11 (November 1985), 1257-1267.

[Bareiss 87] Bareiss, E.R. and Porter, B.W. *A survey of psychological models of concept representation,* The University of Texas at Austin, Technical Report AI TR87-50 (April 1987).

[Barstow 79a] Barstow, D. The roles of knowledge and execution in program synthesis, Proceeding of the 6th International Joint Conference on Artifical Intelligence, Tokyo, Japan (August 1979).

[Barstow 79b] Barstow, D. An experiment in knowledge-based automatic programming, *Artificial Intelligence Journal*. Vol. 12 , No. (August 1979), 73-119.

[Barstow 79c] Barstow, D. *Knowledge-Based Program Construction*, Elsevier-North (1979).

[Barstow 82a] Barstow, D., Duffey, R., Smoliar, S. and Vestal, S. An overview of FNIX, Second National Conference on Artificial Intelligence, Pittsburgh, Pennsylvania (August 1982).

[Barstow 82b] Barstow, D., Duffey, R., Smoliar, S. and Vestal, S. An automatic programming system to support an experimental science, Sixth International Conference on Software Engineering, Tokyo, Japan (September 1982).

[Barstow 84] Barstow, D. A perspective on automatic programming, *AI Magazine*. Vol. 5, No. 1 (Spring 1984), 5-27.

[Barstow 85a] Barstow, D. Domain-specific automatic programming, *IEEE Transactions on Software Engineering*. Vol. SE-11, No. 11 (November 1985), 1321-1336.

[Barstow 85b] Barstow, D., Barth, P., Dietz, R. and Greenspan, S. Observations on specifications and automatic programming, Proceedings of the 3rd International Workshop on Software Specification and Design, London (1985).

[Barstow 89] Barstow, D. Automatic programming for device control software, Workshop on Automating Software Design, Detroit, MI (August 21, 1989), 16-33.

[Batory 87] Batory, D. A mollecular database technology, University of Texas Technical Report (1987).

[Biggerstaff 87] Biggerstaff, T.J. *Hypermedia as a tool to aid large scale reuse, MCC Technical Report, STP-202-87*, (July 1987).

[Biggerstaff 89a] Biggerstaff, T.J., J. Hoskins and D.E. Webster. DESIRE: A System for Design Recovery, Technical Report STP-081-89, MCC (1989).

[Biggerstaff 89b] Biggerstaff, T.J. Design for Maintenance and Reuse, IEEE Computer (1989).

[Birtwistle 73] Birtwistle, G. in *Simula Begin*, Auerbach, Philadelphia, PA (1973).

[Blalock 72] Blalock, H. in *Social Statistics* (eds. Larsen, O.N.), McGraw-Hill, New York (1972).

[Bobrow 83a] Bobrow, D. and Stefik, M. Knowledge programming in loops, *AI Magazine* (August 1983).

[Bobrow 85] Bobrow, D. If Prolog is the answer, what is the question? or what it takes to support AI programming paradigms, *IEEE Transactions on Software Engineering*. Vol. SE 11, No. 11 (November 1985), 1401-1408.

[Bobrow 86] Bobrow, D.G. and Stefik, M. Perspectives on artificial intelligence programming, *Science*. Vol. 231, No. 4741 (February 1986), 951.

[Boehm 76] Boehm, B. Software engineering, *IEEE Transactions on Computers*. Vol. C-25, No. 12 (December 1976), 1226-1241.

[Boehm 84] Boehm, B.W., Gray, T.E. and Seewaldt, T. Prototyping versus specifying: A multiproject experiment, *IEEE Transactions on Software Engineering*. Vol. SE-103, No. 3 (March 1984), 290-302.

[Boehm 88] Boehm, B.W. A spiral model of software development and enhancement, *IEEE Transactions on Computers*. Vol. C-21, No. (May 1988), 61-72.

[Booch 86] Booch, G. Object-oriented development, *IEEE Transactions on Software Engineering* (February 1986), 211-221.

[Borgida 84] Borgida, A., Mylopoulos, J. and Wong, H. "Generalization/specialization as a basis for software specification" in *On Conceptual Modelling* (eds. Brodie, M., John, M. & Schmidt, J.), Springer-Verlag, New York (1984), 87-117.

[Borgida 85a] Borgida, A. Features of languages for the development of information systems at the conceptual level., *IEEE Software*. Vol. 2, No. 1 (January 1985), 63-72.

[Borgida 85b] Borgida, A., Greenspan, S. and Mylopoulos, J. Knowledge representation as the basis for requirements specifications, *IEEE Computer* (April 1985).

[Borning 82] Borning, A. and Ingalls, D. Multiple inheritance in smalltalk-80, Proceedings AAAI (1982), 234-237.

[Brachman 83] Brachman, R.J. What IS-A is and isn't: An analysis of taxonomic links in semantic networks, *IEEE Computer*. (1983), 30-36.

[Brachman 85a] Brachman, R. and Schmolze, J. An overview of the Kl-ONE knowledge representation system, *Cognitive Science*, Vol. 9, No.2 (1985), 171-216.

[Brachman 85b] Brachman, R. I lied about the trees', or, defaults and definitions in knowledge representation, The AI Magazine (Fall 1985), 80-92.

[Brachman 85c] Brachman, R. and Levesque, H. Readings in knowledge representation. Morgan Kaufmann Publishers, Inc., Los Altos, CA (1985).

[Brodie 84] Brodie, M., Mylopoulos, J. and Schmidt, J. in *On Conceptual Modelling*, Springer-Verlag, New York (1984).

[Brodie 86] Brodie, M.L. and Mylopoulos, J. in *On Knowledge Base Management Systems* (eds. 86], [.), Springer-Verlag (1986).

[Brooks 75] Brooks, F.P.J. *The mythical man-month: Essays on software engineering*, Addison-Wesley, Reading, MA (1975).

[Brooks 87] Brooks, F. No silver bullet, *IEEE Computer*. Vol. 20, No. 4 (1987) 10-19.

[Browne 82] Browne, J.C. and Smith, T. An object-oriented, capability-based architecture, Proceedings of 16th IBM Computer Science Symposium (October 1982).

[Browne 84] Browne, J.C., *et al.* Zeus: An object-oriented distributed operating system, Proceedings ACM National Conference (1984).

[Browne 89] Browne, J.C. and Hufnagel, S.P. Performance properties of vertically partitioned object-oriented systems, *IEEE Transactions on Software Engineering*. Vol. 15, No. 8 (August 1989), 935-946.

[Brownlee 65] Brownlee, K.A. in *Statistical Theory and Methodology in Science and Engineering 2nd Edition*, John Wiley and Sons, New York (1965).

[Bruce 86] Bruce, K. and Wegner, P. *Algebraic and lambda calculus models of subtype and inheritance,* Brown Technical Report , (August 1986).

[Bruns 86a] Bruns, G., Bridgeland , D. and Webster, D. Design technology assessment: Gist., MCC Technical Report Number STP-369-86, Austin, TX (1986).

[Bruns 86b] Bruns, G. *Design technology assessments: Affirm, CAEDE, Draco, PNUT,* MCC Technical Report Number STP-197-87 , (1987).

[Bruns 88] Bruns, G. and Potts, C. *Domain modeling approaches to software development,* MCC Technical Report Number STP-186-88 (1988).

[Cardelli 84] Cardelli, L. "A semantics of multiple inheritance" in *Semantics of Data Types, Lecture Notes in Computer Science*, Springer-Verlag (1984), 51-67.

[Cardelli 85] Cardelli, L. and Wegner, P. On understanding types, data abstraction, and polymorphism, *Computing Surveys*. Vol. 17, No. 4 (December 1985), 471-522.

[Caspari 76] Caspari, J.A. Wherefore accounting data - Explanation, predication and decisions, *The Accounting Review*. Vol. LI, No. 4 (1976), 739-745.

[Ceri 89] Ceri, S., Crespi-Reghizzi, S., Maio, A.D. and Lavazza, U.A. Software prototyping by relational techniques: Experiences with program construction systems, *IEEE Transactions on Software Engineering*. Vol. 14, No. 11 (November 1989), 1597-1608.

[Chambers 72] Chambers, R.J. Measurement in current accounting practice: A critique, *The Accounting Review* (July, 1972), 488-509.

[Chambers 73] Chambers, R.J. Accounting principles or accounting policies?, The Journal of Accountancy (May 1973), 48-53.

[Chen 76] Chen, P. The entity-relationship model: Toward a unified view of data, *ACM Transactions on Database Systems*. Vol. 1, No. 1 (March 1976).

[Coad 90] Coad, P. and Yourdon, E. *Object-Oriented Analysis*, Prentice-Hall, Inc. (1990).

[Codd 79] Codd, E.F. Extending the database relational model to capture more meaning, *ACM Transactions on Database Systems*. Vol. 4, No. 4 (December 1979), 397-434.

[Coombs 53] Clyde, H. Theory and methods of social measurement, Research Methods in the Behavioral Sciences, Holt, Reinhart, and Winston, New York (1953), 471-535.

[Coombs 54] Coombs, C.H., Raiffa, H. and Thrall, R.M. Some views on mathematical models and measurement theory, *Psychological Review*. Vol. 61 (March 1954), 132-144.

[Coombs 60] Coombs, C.H. A theory of data, *Psychological Review*. Vol. 67 (1967), 143-159.

[Cox 86] Cox, B.*Object Oriented Programming*, Addison-Wesley, Reading, MA (1986).

[Curtis 80] Curtis, B. Measurement and experimentation in software engineering, *Proceedings of the IEEE*. Vol. 68, No. 9 (September 1980), 1144-1157.

[Curtis 87] Curtis, B., Krasner, H., Shen, V. and Iscoe, N. On building software process models under the lamppost , Proceedings of the 9th International Conference on Software Engineering , Washington, DC (1987), 96-103.

[Curtis 88] Curtis, B., Krasner, H. and Iscoe, N. A field study of the software design process for large systems, *Communications of the ACM*. Vol. 31, No. (November 1988), 1268-1287.

[Dahl 66] Dahl, O.J. and Nygaard, K. SIMULA—an algol based simulation language, *Communications of the ACM* . Vol. 9, No. (1966), 671-678.

[Dahl 72] Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R. Hierarchical program structures, *Structured Programming*, New York (1972), 175-220.

[Danforth 88] Danforth, S. and Tomlinson, C. Type theories and object-oriented programming, *ACM Computing Surveys*, Vol. 20, No. 1 (March 1988).

[Date 81] Date, C.J. *An introduction to data base systems 3rd ed.*, Addison-Wesley Publishing, Reading, MA: (1981).

[Date 84] Date, C.J. "Relational database design" in *A Guide to DB2*, Addison-Wesley Publishing, Reading, MA (1984), 277-293.

[Date 86] Date, C.J. in *Relational Database, Selected Writings*, Addison-Wesley, Reading, MA (1986).

[Davis 88] Davis, A.M., Bersoff, E.H. and Comer, E.R. A strategy for comparing alternative software development life cycle models, *IEEE Transactions on Software Engineering*. Vol. 14, No. 10 (October 1988).

[Deutsch 83a] Deutsch, P. "The dorado smalltalk-80 implementation: Hardware architecture's impact on software architecture" in *Smalltalk-80: Bits of History, Words of Advice* (eds. Krasner, G.), Addison-Wesley, Reading, MA (1983), 113-126.

[Deutsch 83b] Deutsch, P. and Schiffman, A.M. Efficient implementation of the smalltalk-80 system, Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages (1983), 297-302.

[Dijkstra 72] Dijkstra, E. "Notes on structured programming" in *Structured Programming* (eds. Dahl, O.J., Dijkstra, E.W. & Hoare, C.A.R.), Academic Press, New York (1972).

[Everest 77] Everest, G.C. and Weber, R. A relational approach to accounting models, *The Accounting Review*. Vol. LII, No. 2 (1977), 340-359.

[Feather 82] Feather, M.S. A system for assisting program transformation, *ACM Transformations on Programming Languages and Systems*. Vol. 4, No. 1 (January 1982), 1-20.

[Feather 87a] Feather, M.S. Language support for the specificaton and development of computer systems, *ACM transactions on programming languages and systems*, Vol. 9, No. 2 (April 1987), 198-234.

[Feather 87b] Feather, M.S. The evolution of composite system specifications, Proceedings of the Fourth International Workshop on Software Specification and Design (April 1987), 52-57.

[Feather 89] Feather, M.S. Constructing specifications by combining parallel elaborations, *IEEE Transaction on Software Engineering*. Vol. 15, No. 2 (February 1989), 198-208.

[Feller 71] Feller, W. in *An Introduction to Probability Theory and Its Applications, Volume 11*, John Wiley and Sons, New York (1971).

[Feller 50] Feller, W. in *An Introduction to Probability Theory and Its Applications, Volume 1*, John Wiley and Sons, New York (1950).

[Fickas 85] Fickas, S. Automating the transformational development of software, *IEEE Transactions on Software Engineering*. Vol. SE-11, No. 11 (November 1985), 1268-1277.

[Fickas 86] Fickas, S. A knowledge-based approach to specification acquisition and construction, CIS-TR 85-13, Computer Science Department, University of Oregon, Eugene, Oregon (1986).

[Fickas 87] Fickas, S. Automating the analysis process: An example, Fourth International Workshop on Specification and Design (April 1987), 58-67.

[Fickas 88] Fickas, S. and P. Nagarajan. Being suspicious: Critiquing problem specification, Proceedings of the 7th National Conference on Artificial Intelligence (August 1988), 19-24.

[Goguen 88] Goguen, J. and Meseguer, J. "Unifying functional, object-oriented and relational programming with logical semantics" in *Research Directions in Object-Oriented Programming* (eds. Shriver, B. & Wegner, P.), MIT Press, Cambridge, MA (1988), 417-478.

[Goldberg 83] Goldberg, A. and Robson, D. in *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA (1983).

[Goldberg 84] Goldberg, A. in *Smalltalk-80: The Interactive Programming Environment,* Addison-Wesley, Reading, MA (1984).

[Green 75] Green, C. and Barstow, D. Some rules for the automatic synthesis of programs, Proceedings of the Fourth International Joint Conference on Artificial Intelligence (September 1975.), 232-239.

[Green 76] Green, C. The design of the PSI program synthesis system, Second International Conference on Software Engineering (1976).

[Green 78] Green, C. and Barstow, D. On program synthesis knowledge, Artificial Intelligence Journal (November 1978), 241-279.

[Green 79] Green, C. Results in knowledge based program systhesis, 6th International Joint Conference on Artificial Intelligence (1979), 342-344.

[Green 82] Green, C. Knowledge-based programming self-applied, *Machine Intelligence* . Vol. 10, No. (1982).

[Green 83] Green, C., Luckham, D., Balzer, R., Cheatham, T. and Rich, C. Report on a knowledge-based software assistant, RADC-TR-83-195, Rome Air Development Center (1983).

[Green 86] Green, C. and Barstow, D. On program synthesis knowledge, Artificial Intelligence, Vol. 10, No. 3 (November 1978, 1986), 241-279.

[Greenspan 82] Greenspan, S., Borgida, A. and Mylopoulos, J. Capturing more world knowledge in the requirements specification , Proceedings of the 6th International Conference on Software Engineering, Tokyo, Japan (1982).

[Greenspan 84] Greenspan, S. Requirements modeling: A knowledge representation approach to software requirements definition, *Ph.D. Thesis, Computer Science Department* , (1984).

[Greenspan 86] Greenspan, S., Borgida, A. and Mylopoulos, J. A requirements modeling language and its logic, *Information Systems*. Vol. 11, No. 1 (1986), 9-23.

[Guttag 77] Guttag, J. Abstract data types and the development of data structures, *Communications of the ACM* (June 1977), 396-404.

[Guttag 85] Guttag, J., Horning, J. and Wing, J. The larch family of specification languages, *IEEE Software*. Vol. 2, No. 5 (September 1985), 24-36.

[Haase 82] Haase, V. and Koch, G. Application-oriented specifications, *IEEE Computer Magazine (*May 1982), 10-60.

[Hirschheim 89] Hirschheim, R. and Klein, H.K. Four paradigms of information systems development, *Communications of the ACM*. Vol. 32, No. 10 (October 1989), 1199-1216.

[Hoare 72] Hoare, C.A.R. Proof of correctness of data representations, *Acta Informatica.* Vol. 1, No. (1972), 271-281.

[Hoare 73] Hoare, C.A.R. Hints on programming language design, *Computer Science Department Report* , (Oct ober 1973), 27.

[Hoare 74] Hoare, C.A.R. Monitors: An operating system structuring concept, *Communications of the ACM*. Vol. 17, No. 10 (Oct ober 1974), 549-558.

[Hoare 85] Hoare, C.A.R. in *Communicating Sequential Processes*, Prentice Hall (1985).

[Hufnagel 89] Hufnagel, S.P. and Browne, J.C. Performance properties of vertically partitioned object-oriented systems, *IEEE Transactions on Software Engineering*. Vol. 15, No. 8 (August 1989).

[Hull 87] Hull, R. and King, R. Semantic database modeling: Survey, applications, and research issues, *ACM Computing Surveys*. Vol. 19, No. 3 (September 1987), 201-260.

[Ingalls 78] Ingalls, D.H. The smalltalk-76 programming system: Design and implementation, *Symposium on Principles of Programming Languages*, Tucson, AZ (January 1978), 9-16.

[Ingalls 83] Ingalls, D.H. "The evolution of the smalltalk virtual machine" in *Smalltalk-80, Bits of History, Words of Advice* (1983).

[Ingalls 86] Ingalls, D.H. A simple technique for handling multiple polymorphism, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon (November 1986), 347-360.

[Iscoe 86] Iscoe, N. CRTForm: An object-oriented application development system using type and type-type managers, *Masters Report, Dept. of Computer Sciences University of Texas* (August 1986).

[Iscoe 88a] Iscoe, N. A database configuration system that learns while it optimizes , Proceedings of the Third Annual Rocky Mountain Conference On Artificial Intelligence, Denver, CO (June 1988.).

[Iscoe 88b] Iscoe, N. Domain-specific reuse: An object-oriented and knowledge-based approach, software reuse: Emerging technology, IEEE Computer Society, Washington, DC (September 1988), 299-308.

[Jackson 83] Jackson, M. *Software development*, Prentice-Hall (1983).

[Jalote 89] Jalote, P. Testing the completeness of specifications, *IEEE Transactions on Software Engineering*. Vol. 15, No. 5 (May 1989), 526-531.

[Kahn 81] Kahn, K. iMAX: A multiprocessor operating system for an object-based computer, Eighth Symposium on Operating Systems Principles (1981), 12-36.

[Kant 79] Kant, E. A knowledge-based approach to using efficiency estimation in program synthesis, 6th International Joint Conference on Artificial Intelligence (1979), 457-462.

[Kant 81a] Kant, E. and Barstow, D. The refinement paradigm: the interaction of coding and efficiency knowledge in program synthesis", IEEE Transactions on Software Engineering (September 1981), 458-471.

[Kant 81b] Kant, E. in *Efficiency in Program Synthesis*, UMI Research Press (1981).

[Kant 83] Kant, E. On the efficient synthesis of efficient programs, *Artificial Intelligence Journal* (1983).

[Kant 85] Kant, E. Understanding and automating algorithm design, IEEE Transactions on Software Engineering (November 1985).

[Keeney 76] Keeney, R. and Raiffa, H. *Decisions with multiple objectives: Preferences and value tradeoffs*, John Wiley and Sons (1976).

[Kemmerer 85] Kemmerer, R. Testing formal specifications to detect design errors, *IEEE Transactions on Software Engineering*. Vol. 11, No. 1 (January, 1985), 32-43.

[Kent 78] Kent, W. *Data and reality*, Elsevier North-Holland, New York, 1978.

[Kish 65] Kish, Leslie. *Survey Sampling*, John Wiley & Sons, Inc. New York (1965).

[Knuth 68] Chapter 1 p. 16 Knuth, D.E. Semantics of context-free languages, *Mathematical Systems Theory*. Vol. 2, No. 2 (June 1968), 127-145.

[Kuhn 70] Kuhn, T. in *The Structure of Scientific Revolutions*, University of Chicago Press (1970).

[Labovitz 70] Labovitz, S. Statistical usage in sociology: Sacred cows and ritual, *Sociological Methods and Research*. Vol. 1, No. (1972), 13-38.

[Labovitz 72] Labovitz, S. The assignment of numbers to rank order categories, *American Sociological Review*. Vol. 35, No. (1970), 515-524.

[Lenat 88a] Lenat, D. and Guha, R.V. *MCC Technical Report ACA-AI-300-88, The world according to CYC* (September 1988).

[Lenat 88b] Lenat, D., Guha, R.V. and Wallace, D.V. *MCC Technical Report No. ACA-AI-302-88,* The CycL representation language (September 1988).

[Lenat 89] Lenat, D.B. Ontological versus knowledge engineering, *IEEE Transactions on Knowledge and Data Engineering*. Vol. 1, No. 1 (March 1989), 84-88.

[Lieberman 81] Lieberman, H. *A preview of act 1,* Massachusetts Institute of Technology, Artifical Intelligence Laboratory Memo, (June 1981).

[Liskov 75] Liskov, B. and Zilles, S. Specification techniques for data abstractions, *IEEE Transactions on Software Engineering*. Vol. 1, No. 1 (March 1975), 7-19.

[Lubars 86] Lubars, M.D. and Harand i, M.T. Intelligent support for software specification and design, *IEEE Expert*. Vol. 1, No. 4 (Winter 1986), 33-41.

[Lubars 88] Lubars, M.D. The knowledge-based refinement paradigm and IDeA: Concepts, limitations, and future directions, Proceedings of the Workshop on Automating Software Design, St. Paul, MN (August 1988), 68-73.

[Lubars 88] Lubars, M.d. The IDeA design environment, Proceedings of the Eleventh International Conference on Software Engineering, Pittsburgh, PA (May 1989), 23-32.

[Lubars 89] Lubars, M.D. Software design support in the ROSE project, Proceedings of the Workshop on Automating Software Design IJCAI-89, Detroit, MI (August 1989), 176-179.

[Lugi 89] Lugi. Software evolution through rapid prototyping, *Computer*. Vol. 22, No. 5 (May 1989), 13-27.

[Maier 83] Maier, D. *The theory of relational databases*, Computer Science Press, Inc., Rockville, MD (1983).

[McCracken 81] McCracken, D. and Jackson, M. "A minority dissenting position" in *Systems Analysis and Design -- A Foundation for the 1980's* (eds. W.W. Cotterman et al., e.), Elsevier North-Holland , New York (1981), 551-553.

[McCullough 83] McCullough. Vol. No.

[Meyer 86] Meyer, B. Genericity versus inheritance, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, OR (November 1986), 391-405.

[Meyer 88] Meyer, B.*Object-Oriented Software Construction*, Prentice Hall, NJ (1988).

[Moon 86] Moon, D.A. Object-oriented programming with flavors, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, OR (November 1986).

[Mostow 85] Mostow, J. Foreword: What is AI and what does it have to do with software engineering? *IEEE Transactions on Software Engineering*. Vol. SE-11, No. 11 (November 1985), 1253-1256.

[Mylopoulous 84] Mylopoulous, J. and Levesque, H. "An overview of knowledge representation" in *On Conceptual Modelling* (eds. Brodie & Myolopoulous), Springer Verlag, NY (1984), 3-17.

[Neighbors 80] Neighbors, J.M. *Software construction using components*, Ph.D. Thesis, University of California-Irvine (1980).

[Neighbors 84a] Neighbors, J.M. The draco approach to constructing software from reusable components, *IEEE Transactions on Software Engineering*. Vol. 10, No. 5 (September 1984), 564-574.

[Neighbors 84b] Neighbors, J.M., Arango, G. and Leite, J.C. *Draco 1.3 users manual*, Vol. No. (September 1984).

[Peckham 88] Peckham, J. and Maryanski, F. Semantic data models, *ACM Computing Surveys*. Vol. 20, No. 3 (1988), 153-189.

[Pollack 81] Pollack, F.J., Kahn, K.C. and Wilkinson, R.M. *The iMAX-432 object filing system*, Eighth Symposium on Operating Systems Principles (1981), 137-147.

[Reubenstein 89] Reubenstein, H.B. and Waters, R.C. *The requirements apprentice: An initial scenario*, Proceedings of the Fifth International Workshop on Software Specification and Design, Pittsburgh, PA (May 19-20, 1989), 211-21.

[Rich 78] Rich, C. and Shrobe, H. Initial report on a LISP programmer's apprentice, *IEEE Transactions on Software Engineering*. Vol. 4, No. 6 (November 1987).

[Rich 82] Rich, C. *Knowledge representation languages and predicate calculus: How to have your cake and eat it too*, Proceedings of the AAAI National Conference, Pittsburgh, Pennsylvania (August 1982).

[Rich 87] Rich, C., Waters, R.C. and Reubenstein, H.B. *Toward a requirements apprentice*, Proceedings of the Fourth International Workshop on Software Specification and Design (April 1987), 79-86.

[Rich 88a] Rich, C. Automatic programming: Myths and prospects, *Computer* (August 1988), 40-51.

[Rich 88b] Rich, C. and Waters, R.C. The programmer's apprentice: A research overview, *Computer*. Vol. 21, No. 11 (November 1988), 11-25.

[Roberts 79] Roberts, F.S. in *Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences*, Addison-Wesley, Reading, MA (1979).

[Robinson 89] Robinson, W.N. *Integrating multiple specifications using domain goals,* Proceedings of the Fifth International Workshop on Software Specification and Design, Pittsburgh, PA (May 19-20, 1989), 219-226.

[Rosenberg 68] Rosenberg, M. in *The Logic of Survey Analysis*, Basic Books, Inc., New York (1968).

[Ross 77] Ross, D. Structured analysis (SA): A language for communicating ideas, *IEEE Transactions on Software Engineering* (January 1977).

[Royce 70] Royce, W. *Managing the development of large software systems: Concepts and techniques*, Proceedings WESCON (August 1970).

[Schwartz 77] Schwartz, S.P. *Naming, Necessity, and Natural Kinds*, Cornell University Press, London (1977).

[Scott 76] Scott, D.S. Data types as lattices, *SIAM J. Comput.* Vol. 5, No. 3 (1976), 523-587.

[Shaw 84] Shaw, M. "The impact of modeling and abstraction concerns on modern programming languages" in *On Conceptual Modeling* (eds. Brodie, M.L. & Mylopoulos, J.), Springer-Verlag, New York (1984), 49-83.

[Shlaer 88] Shlaer, S. and Mellor, S.J. in *Object-Oriented Systems Analysis Modeling the World in Data*, Yourdon Press Computing Series, Englewood Cliffs, New Jersey (1988).

[Shortley 73] Shortley, G. and Williams, D. in *Elements of Physics*, Prentice Hall (1973).

[Smith 77] Smith, J. and Smith, D. *Database abstractions: Aggregation and generalization* , ACM Transactions on Database Systems (1977), 1278-1295.

[Smith 85] Smith, D., Kotik, G. and Westfold, S. Research on knowledge-based software environments at Kestrel Institute, *IEEE Transactions on Software Engineering*. Vol. SE-11, No. 11 (November 1985).

[Smoliar 83] Somliar, S. and Barstow, D. Who needs languages, and why do they need them? or no matter high the level it's still programming, SIGPLAN: Symposium on Programming Language Issues in Software Systems, San Francisco, California (June 1983).

[Snyder 86] Snyder, A. Encapsulation and inheritance in object-oriented programming languages, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, OR (November 1986), 38-45.

[Snyder 88] Snyder, A. "Inheritance and the development of encapsulated software systems" in *Research Directions in Object-Oriented Programming* (ed. Wegner, S.A.), MIT Press, Cambridge, MA (1988), 165-188.

[Stamper 73] Stamper, R. *Information in business and administrative systems*, John Wiley and Sons, New York (1973).

[Steensgaard 89] Steensgaard-Madsen, J. Typed representation of objects by functions, Transactions on Programming Languages and Systems (January 1989), 67-89.

[Stefik 85] Stefik, M.J. and Bobrow, G. Object-oriented programming: Themes and variations, *The AI Magazine* (Winter 1985), 40-61.

[Stefik 86] Stefik, M.J., Bobrow, D. and Kahn, K. Integrating access-oriented programming into a multiparadigm environment, IEEE Software (January 1986), 10-18.

[Stevens 46] Stevens, S.S. On the theory of scales of measurement, *Science 103 (*1946), 677-680.

[Stevens 74] Stevens, S.S. "Measurement" in *Scaling A Sourcebook for Behavioral Scientists* (eds. Marnell, G.M.), Aldine, Chicago, IL (1974), 22-41.

[Stonebraker 82] Stonebraker, M. A rules system for a realtional database system, Proceedings of the 2nd International Conference on Databases, Jerusalem, Israel (June 1982).

[Suppes 58] Suppes, P. and Scott, D. Foundational aspects of theories of measurement, *Journal of Symbolic logic*, Vol. 23 (1958), 113-128.

[Swartout 82] Swartout, W. and Balzer, R. An inevitable intertwining of specification and implementation, *Communications of the ACM*. Vol. 25, No. 7 (July 1982), 438-446.

[Teorey 86] Teorey, T.J., Yang, D. and Fry, J.P. A logical design methodology for relational databases using the extended entity-relationship model, *ACM Comput Surv*. Vol. 18, No. 2 (June 1986), 197-222.

[Torgerson 58] Torgerson, W. *Theory and methods of scaling* , John Wiley & Sons (1958).

[Touretzky 86] Touretzky, D.S. *The Mathematics of Inheritance Systems*, Morgan Kaufmann Publishers, Inc. (1986).

[Tsichritzis 82] Tsichritzis, D.C. and Lochovsky, F.H. *Data Models*, Prentice-Hall, Englewood Cliffs, N. J. (1982).

[Tukey 77] Tukey, J.*Exploratory Data Analysis*, Addison Wesley, Reading, MA (1977).

[Ullmann 80] Ullmann, J.D. *Priciples of database systems*, Computer Science Press, Rockville, MD (1980).

[Waters 82] Waters, R. The programmer's apprentice: Knowledge based program editing, *IEEE Transactions on Software Engineering*. Vol. SE-11, No. 11 (January 1982), 1296-1320.

[Waters 85] Waters, R. The programmer's apprentice: A session with KBEmacs, *IEEE Transactions on Software Engineering*. Vol. SE-11, No. 11 (November 1985).

[Webster 88] Webster, D.E. Mapping the design information representation terrain, *Computer*. Vol. 21, No. 12 (December 1988), 8-23.

[Wegner 84] Wegner, P. Capital intensive software technology -- Part 2: Programming in the large , *IEEE Transactions on Software Engineering*. Vol. 1, No. 3 (July 1984), 24-32.

[Wegner 88] Wegner, P. "The object-oriented classification paradigm" in *Research Directions in Object-Oriented Programming* (eds. Wegner, S.a.), MIT Press, Cambridge, MA (1988), 479-560.

[Weitzel 89] Weitzel, J.R. and Kerschberg, L. Developing knowledge-based systems: Reorganizing the system development life cycle, *Communications of the ACM*. Vol. 32, No. 4 (January 1989), 482-488.

[Westfold 84] Westfold, S. Very-high level programming of knowledge-based schemes, Fourth National Conference of the American Association for Artificial Intelligence (August 1984).

[Wile 83] Wile, D. Program developments: Formal explanations of implementations, *Communications of the ACM* (November), 191-200.

[Williams 89] Williams, G.B., Mui, C., Alagappan, V. and Johnson, B.B. Software design issues: A very large information systems perspective, Proceedings of the Fifth International Workshop on Software Specification and Design, Pittsburgh, PA (May 19-20, 1989), 238-241.

[Wills 87] Chapter 1 p. 16 Wills, L.M. *Automated program recognition* MIT Artificial Intelligence Laboratory, Technical Report 904 (February 1987).

[Wing 87] Wing, J. A larch specification of the Library problem, Proceedings of the Fourth International Workshop on Software Specification and Design (April 1987), 34-41.

[Wing 88] Wing, J.*Study of twelve specifications of the library problem* Technical Report, Department of Computer Sciences, Carnegie Mellon University (1988).

[Winograd 79] Winograd, T. Beyond programming languages, *Communications of the ACM*. Vol. 22, No. 7 (July 1979), 391-401.

[Wulf 74] Wulf, W.A. Hydra: The kernel of a multiprocessor operating system, *Communications of the ACM*. Vol. 17, No. 6 (June 1974), 337-345.

[Wulf 75] Wulf, W.A. Vol. No.

[Wulf 76] Wulf, W.A., London, R.L. and Shaw, M. *Abstraction and verification in ALPHARD*, Dept of Computer Science, Carnegie-Mellon University (June 1976).

[Wulf 81] Wulf, W.A., Levin, R. and Pierson, C. in *Hydra/C.mmp*, McGraw-Hill (1981).

[Yourdon 79] Yourdon, E. *Managing the structural techniques*, Prentice-Hall, Inc., NJ (1979).

[Zave 82] Zave, P. An operational approach to requirements specification for embedded systems, *IEEE Transactions on Software Engineering*. Vol. SE-8, No. 3 (May 1982), 250-269.

[Zave 84] Zave, P. The operational versus the conventional approach to software development, *Communications. of ACM*. Vol. 27, No. 2 (February 1984).

[Zave 86] Zave, P. and Schell, W. Salient features of an executable specification language and its environment, *IEEE Transactions on Software Engineering*. Vol. SE-12, No. 2 (February 1986), 312-325.

[Zave 89] Zave, P. A compositional approach to multiparadigm programming, *IEEE Software* (September 1989), 15-25.

[Zilles 84] Zilles, S. "Types, algebras, and modelling" in *On Conceptual Modelling* (eds. Brodie & Mylopoulos.), Springer-Verlag, NY (1984), 441-450.