# GENERATING DOMAIN MODELS FOR PROGRAM SPECIFICATION AND GENERATION

Neil Iscoe, J. C. Browne, John Werth

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

TR-89-13 May 1989

## Abstract

This paper describes a formalized domain modeling technique which we have designed to represent relevant portions of the domain knowledge required to specify and implement application program generators. Successful application generators are always domain specific. Software engineering of application generators and meta-generators must be based on a characterization of domain knowledge which captures the relevant application domain knowledge necessary for the paradigm. The operational 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. The focus is on a representation technique to support the meta-level aspects of such a system—a generator for narrow domain application program generators. A domain model generated using this technique is based on classes that are composed of attributes defined in formal terms that characterize an application domain. Unique to this technique is the integration of domain-specific properties such as units, quantities, granularity, qualitative scales, population parameters into a coherent system of reusable attributes and classes. A system prototype, Ozym, has been developed to experiment with domain model generation.

**Keywords**: Software Engineering, Domain Modeling, Program Generation, Reuse, Object-Oriented, Requirements Elicitation, Specification Techniques

ARPANET: iscoe@cs.utexas.edu

## 1. Introduction

This paper describes a formalized domain modeling technique which we have designed to represent relevant portions of the domain knowledge required to specify and implement application program generators. Successful application generators are always domain specific. Software engineering of application generators and meta-generators must be based on a characterization of domain knowledge which captures the relevant application domain knowledge necessary for the paradigm. The operational 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. Within the context of "automatic program generation," this is what Rich and Waters [1988] call a "narrow domain approach." Our focus is on the model representation to support the meta-level aspects of such a system—a generator for narrow domain application program generators.

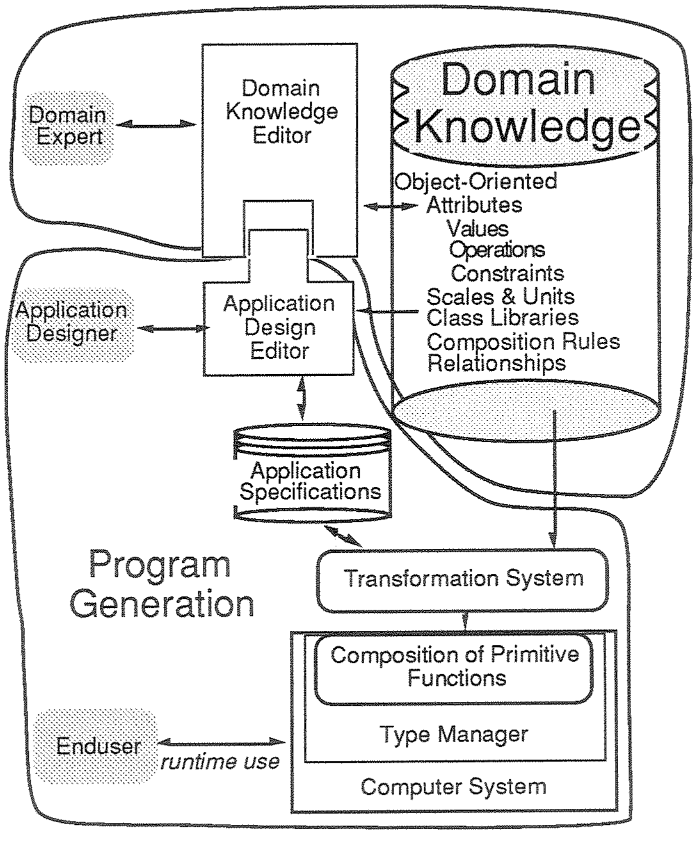


**Figure 1 — Solution Paradigm**

Figure 1 is a diagram of the paradigm. The top right portion of this figure illustrates a conceptual structure for domain knowledge that can be used as the basis for generating domain-specific application program generators. In our view, a domain model generator need be both operationally useable and precise. The representation must be able to be instantiated into a particular application area domain model by a domain expert and then used by application designers to create application programs. Operationally, the representation should be powerful enough to capture the elements that constitute application domain knowledge and general enough to instantiate in several different domains.

A system prototype, Ozym, has been developed to experiment with various aspects of the modeling technique. In terms of Figure 1, Ozym is an implementation of the knowledge base, the domain knowledge editor used by the domain expert, and the application design editor used by the application designer. Ozym is being used to iteratively refine the modeling technique, and consequent domain models.

The lower portion of figure 1 represents a domain-specific transformational program generation system. Application designers, who range in expertise from domain experts to endusers, interact with a design editor to create a set of application specifications. These specifications are then transformed into a composition of primitive functions that can be executed by a type manager. The creation of executable programs by performing a series of transformations on application specifications will not be discussed in this paper. A description of other aspects of the paradigm is given in [Iscoe 88].

## 1.1. Overview of the Modeling Technique

Construction of a domain model begins with the identification of attributes that are common across an application domain, and the subsequent parameterization of the attributes by their characteristics. Classes are then specified in terms of those attributes. Attributes will be formally defined in section 3, and can be considered to be based on scales [Stevens 46], units, granularity, defaults, and population parameters. These attribute properties are used to constrain attribute values, functional transformations of those values, and enduser data values. The unique contribution of this research is the integration of these domain specific attribute properties into a formal attribute and class based representation technique for domain knowledge. Classes are defined by their attributes, their population parameters, and by specialized functions. Libraries of units, derived units, attributes, and classes are organized into hierarchies through the use of inheritance. Classes can be combined through the use of composition operators.

The domain models generated are object-oriented. 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.* Our modeling technique attempts to define and separate out the real world aspects of an application domain. Attributes and classes provide templates for the construction of objects that exist within the runtime life of any particular application domain. Figure 2 is an informal view that precedes the more formal definitions of attribute, class, scale, unit, inheritance, and composition that will be

1 [Dahl 66], [Goldberg 83], [Jackson 83], , [Stefik 85], [Booch 86] [Cox 86], and [Meyer 88] are some of the frequently referenced definitions. Numerous others exist.

detailed throughout the remainder of this paper. It shows the concept of libraries of reusable domain definitions in terms of classes composed of attributes defined in terms of scales and units. Single and multiple inheritance are used to structure domain knowledge and to create new classes.

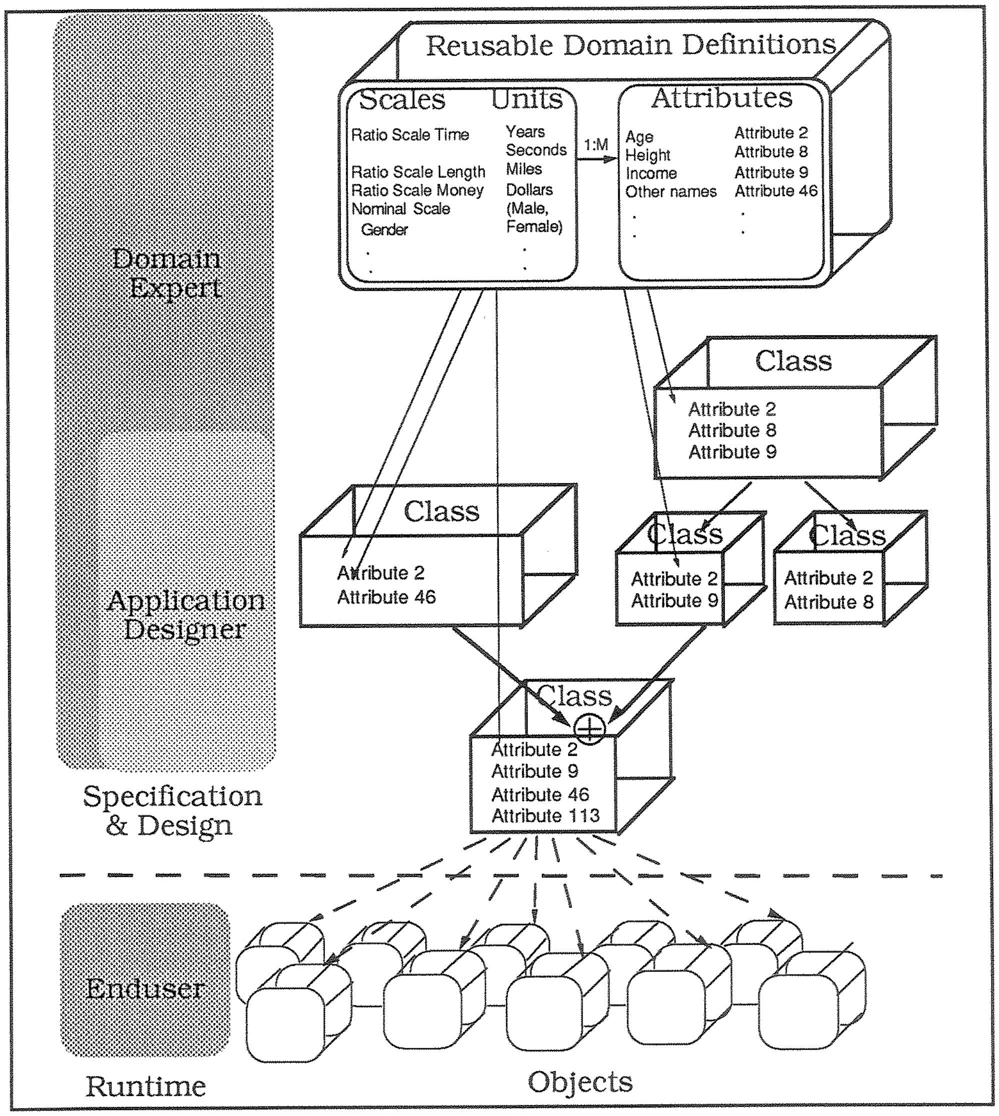


**Figure 2 — Top Level View of A Domain Model**

This paper begins with a brief discussion of application domain knowledge and the paradigm for application programming that underlies this research. The modeling methodology and portions of a domain model are described with examples drawn from the library[2] and other well known problems. This approach to the representation of domain knowledge synthesizes concepts and techniques from artificial intelligence, software engineering, type theory, databases, object-oriented systems, and statistics to attain its goals.

[2] The classic specification problem for a simple library published by [Kemmerer 85]. [Wing 88] analyzes twelve views oriented towards program specification, and gives a history of the problem and its solutions.

## 2. Attributes

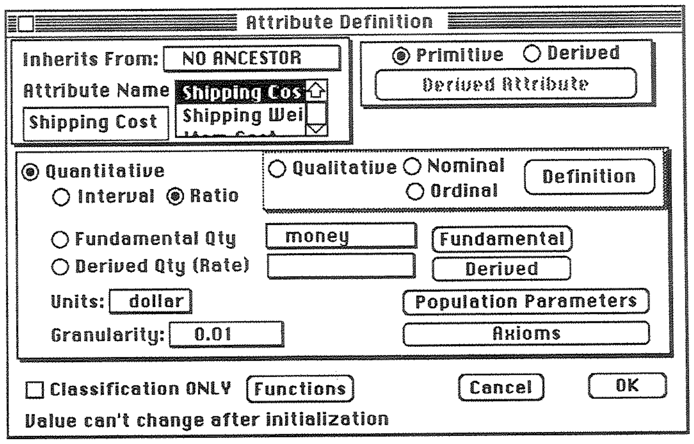
Attributes are building blocks for classes and are key components of domain models. In most programming and specification languages, values are classified in terms of basic types like boolean, integer, real number, enumerated, and others that have been mapped up from computer architectures. In contrast, a domain model represents information that ordinarily is not maintained throughout the development and runtime lifecycle of a program. A domain model begins with value sets, operations, and constraints imposed by the application domain, and then works towards a machine representation. Attributes provide the first part of a definition for this type of application knowledge. In addition to providing a basis for the model representation, we believe that the approach developed in this section can be useful in other software engineering specification and design methodologies.

In some senses, attributes can be considered to be abstract data types ([Liskov 74], [Guttag 77], [Danforth 88] that encapsulate information about the properties of classes. Attributes are primitives that are instantiated from domain measures expressed in terms of scales, units, and granularity. The resultant value sets are characterized and constrained by domain-specific population parameters that are also used to infer default information. Axioms add additional information and further restrict the set of attribute operations. The following definition will be further developed throughout the rest of this section.

An attribute, A, consists of :
- a unique name $\mathcal{N}(A)$
- a measurement scale $\mathcal{M}(A)$ and (when appropriate) measurement unit & granularity
- a set of population parameters $\mathcal{PP}(A)$
- a set of values $\mathcal{V}(A)$
- a set of operations $\mathcal{F}(A)$, such that if $f \in \mathcal{F}(A)$ then $f: \mathcal{V}(A) \rightarrow \mathcal{V}(A)$
- an initialization function i, $i: \mathit{Null} \rightarrow \mathcal{V}(A)$
- a set of axioms or integrity constraints $\mathcal{X}(A)$.

**Attribute Definition**

Figure 3 is a Macintosh screen print that shows how the Ozym implementation captures information about an attribute. Throughout the remainder of this paper, the formal modeling technique will sometimes be illustrated in terms of the implementation. The complete formal description will not be presented in this paper, but can be found in [Iscoe 89].


Attribute Definition

Inherits From: NO ANCESTOR

Attribute Name: Shipping Cost (Shipping Cos, Shipping Wei)

[✓ Primitive] Derived

Derived Attribute

[✓ Quantitative] Interval [✓ Ratio]

Qualitative Nominal Ordinal

Definition

Fundamental Qty: money

Derived Qty (Rate)

Fundamental

Derived

Units: dollar

Granularity: 0.01

Population Parameters

Axioms

Classification ONLY

Functions

Cancel

OK

Value can't change after initialization

**Figure 3 — Ozym Attribute Definition (Main Screen)**

## 2.1. *Value Sets $\mathcal{V}(A)$ and Operations on Value Sets $\mathcal{F}(A)$*

A domain model attempts to represent the natural value restrictions of an application domain instead of relying on the more basic data types of strongly typed specification and programming languages. Consideration of the general class of state transformation functions, ($f : V \rightarrow V$, on the value set, $V = \mathcal{V}(A)$, of a single attribute, A, of any object O), is a useful abstraction, but it does not adequately constrain the potential number of such functions[3], which is $|V|^{|V|}$. Domain knowledge, such as scale type, allows selection of certain appropriate functions, the $\mathcal{F}(A)$, while also facilitating exclusion of other inappropriate functions. The next section discusses the scales that are used to define, specify, and construct the characteristics of attribute value sets, V, and the functions, $f: V \rightarrow V$, that perform mappings of these value sets.

## 2.2. *Scaling Theory*

Scaling theory is a methodology with a mathematical basis that is used for classification in a variety of fields. It was developed as a system for mapping properties of objects and events to numerical scales with well defined properties. Scales are used in a domain model as a means for classifying object attributes in a manner that is formal and yet captures enough of the underlying real world semantics that the resultant domain model can be used for program specification and transformational purposes. Seminal papers on scaling theory include [Stevens 46], [Torgerson 58], and [Coombs 60]. Scales have not been previously used for these software engineering purposes, but [Curtis 80] points out the value of their use for software engineering measurement. The classical scale types are nominal, ordinal, interval, and ratio, and are briefly described in the following four sections.

### Nominal scales

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

3 The domain and the range, V, are the same. Consequently, |D|**|R|, becomes |V|**|V|.

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

**Nominal Scale Definition**

Domain semantics 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. A nominal scale to classify children's coloring book covers might be the colors red, yellow, green, and blue. The following definition shows how nominal scales map cleanly to the notion of enumerated type:

```
Colors : Nominal_scale (Red, Yellow, Green, Blue)
    Airplane_book_cover.Color := Blue;
    Firetruck_book_cover.Color := Red
```

## Ordinal Scales

*Nominal Scales* are used when there is no information about items other than the category to which they belong. *Ordinal Scales* are nominal scales for which a strict ordering between categories can be obtained by ranking the groups according to the relative amounts that they possess of some characteristic defined by the scale[4].

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

**Ordinal Scale Definition**

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 color example, colors can be placed on an ordinal scale of perceived warmth as follows:

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

The following are some of the interesting functions on an ordinal scale value set V. (These definitions require the addition of a special item called *error* [5] to all of the value sets in a domain model.)

*Identity* {f | f(x) = x}

*Monotonically Increasing* {f | (f(x) << f(y) <--> x <<y) or (f(x) = error) }

*Increment* {f | (f(x) = succ(x) or (f(x) = error)}

*Monotonically Decreasing* {f | (f(x) >> f(y) <--> x >> y ) or (f(x) = error)

*Decrement* {f | (f(x) = pred(x) or (f(x) = error)}

4 [Coombs 53] also describes partial orders and other scales for which the conditions of transitivity and asymmetry are violated.

5 More formally, For all attributes i, there exists an element *error* such that $error \in \mathcal{V}(A)$, and that f(*error*) = *error* for every $f \in \mathcal{F}(A)$,

## Units, Quantities, and Granularity

Nominal and ordinal scales are often called qualitative measures because they are not measuring amounts or quantities of units. The following definition begins the discussion required to introduce the more quantitative interval and ratio scales.

A Quantity is defined in terms of:

- a unit that is the finest grained measure normally used in a domain.
- a measurement granularity, that is the highest degree of precision to which a unit is normally expressed within a domain.

**Quantity Definition**

The selection of a unit is a domain-specific decision that is based on both tradition and technology. Time, for example, is measured in minutes for long distance telephone calls, seconds for football games, hundredths of seconds for Olympic events, nanoseconds for computer chips, and millennia for geologic transformations. Cost in dollars, distance in miles, weight in pounds, speed in miles/hour, salary in dollars/hour, salary in dollars/year, and salary in dollars/academic-year are all examples of the unit portion of a quantity.

Effects of a domain unit choice can have many consequences. For example, age is usually measured in years. But a year unit granularity can fail 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 years doesn't take into account that ages of:

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

Using an age derived from birthdate where the measurement granularity of the base time unit is day, it is possible to calculate ages based on days, weeks, months, or years. Note, however, that it would be incorrect to calculate age to a finer granularity than day. Furthermore, simply producing the number of days that a person has been alive is not generally the most useful output from a model. Unit quantities are a key component of application domain knowledge.

*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. 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 (tenths of an hour), while in other offices hours are measured in four sub units (quarter hours). 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, and .30 or .40 hours in a tenth of an hour system. Even in systems that are not automated, the choice of granularity can have major effects on revenues, and a client's perception of correctness.

## Interval Scales

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

> An interval scale is a scale that has a unit, u, and assigns a unique multiple (called the magnitude) of the measurement granularity, G(u), of this unit to each category.

**Interval Scale Definition**

These magnitudes impose a strict ordering, hence an interval scale is an ordinal scale that gains additional properties such as addition and subtraction in terms of the scale units. 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.
```

In addition to the functions defined for ordinal scales, the following functions are applicable for an interval scaled value set V :

> *Arbitrary increment* $\{f \mid (f(x) = x + i \text{ and } (i \in V) \text{ and } (i > 0)) \text{ or } (f(x) = \text{error})\}$
>
> *Arbitrary decrement* $\{f \mid (f(x) = x - i \text{ and } (i \in V) \text{ and } (i > 0)) \text{ or } (f(x) = \text{error})\}$

## Ratio Scales

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

**Ratio Scale Definition**

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. Because the zero point in an interval scale, such as temperature in Fahrenheit degrees, is arbitrary it is impossible to say that a Fahrenheit temperature X is twice as hot as a Fahrenheit temperature Y, or to make any other type of ratio comparisons. *Ratio scales* have non-arbitrary zero points, so multiplication and division that have meaning within a domain can be performed on these scales.

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 and years are two different unit intervals superimposed on the same scale. Since they are not even multiples of each other, complicated systems are sometimes used to convert between units of the scale.

Although date is an interval scale, human age is a ratio scale measured in unit years that is derived from the difference between two points of an interval scale with unit day. Since age is a ratio scale, it is possible to make proportional comparisons between the ages of two people. However, it is impossible to make these same type of comparisons between two dates on the a calendar based interval scale.

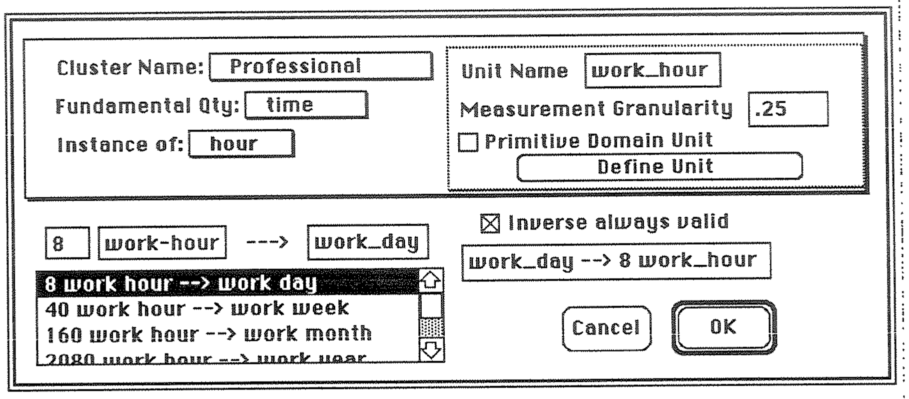
## Fundamental Units and Quantities

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 Kelvin, 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 money in unit dollars is used within the class of United States business application domain models. 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[6]. However, from a pragmatic standpoint it does appear that certain information appears regularly across classes of domains. Ozym stores these fundamental scales, units, and attributes so that they can be used as a basis for developing domain models.

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



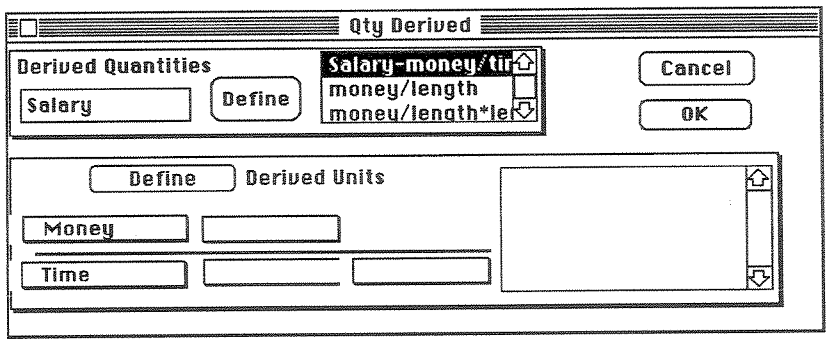
Cluster Name: Professional
Fundamental Qty: time
Instance of: hour
Unit Name work_hour
Measurement Granularity .25
☐ Primitive Domain Unit
8 work-hour ---> work_day
☒ Inverse always valid
work_day --> 8 work_hour

8 work hour --> work day
40 work hour --> work week
160 work hour --> work month
2080 work hour --> work year

**Figure 4 — Defining a Work-hour**

## Derived Quantities

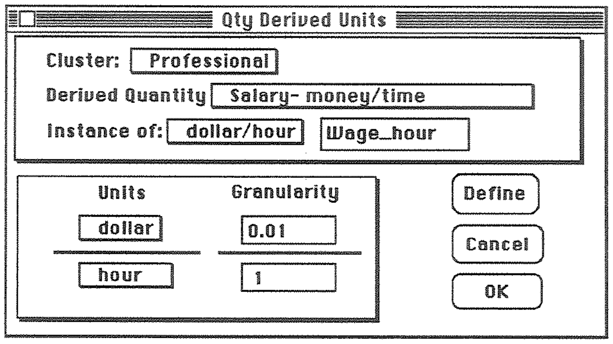
Derived quantities are expressed in terms of *fundamental quantities,* and their dimensions are specified in terms of the powers of the fundamental quantities that were used to create the particular measure. In a domain model, these dimensions are the basis for consistency checks; just as in the physical sciences, a necessary condition for the truth of any equation is that the physical dimensions of each term be identical and that they make sense. For example $/unit time, $/unit_length, $/ (unit_length)$^2$, $/ (unit_length)$^3$, are all reasonable unit combinations in an accounts receivables system but $/(unit_length)$^4$ or $$^2$/unit_length are meaningless combinations of quantities.



Derived Quantities
Salary
Salary-money/tir
money/length
money/length*le
Money
Time

**Figure 5 — Defining the derived quantity Salary (time/money)**

Although unit checking is so obvious that it's rarely stated, it is frequently ignored at the implementation level of an application program. A domain model enforces these restrictions.


Qty Derived Units

Cluster: Professional

Derived Quantity: Salary- money/time

Instance of: dollar/hour | Wage_hour

| Units | Granularity |
|---|---|
| dollar | 0.01 |
| hour | 1 |

**Figure 6 — Dollar/hour is one instance of salary**

## 2.3. *Summary of Scale Types*

Scaling theory has been introduced as a well defined system for constraining values. Nominal scales categorize items and are the most basic scale type. Ordinal scales are nominal scales for which order between categories can be obtained. Interval scales are ordinal scales with well defined units, and ratio scales are interval scales with a real zero point. Scales can also be viewed from the framework of inheritance as gaining additional statistical and transformational operations as they are specialized down an inheritance tree as shown in Figure 7.

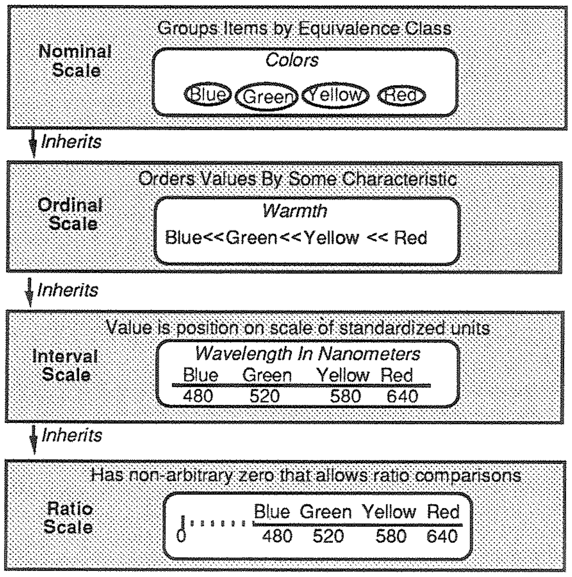


**Figure 7— Scale summary**

## 2.4. *Axioms $\mathcal{X}(A)$*

Axioms place additional restrictions on $\mathcal{V}$(A) and $\mathcal{F}$(A). One major type of restriction is the statement that an attribute is immutable, such as when it is part of the name of an object, or when it is used for classification purposes, as is the case with gender or book_type.

A is an immutable attribute if $\mathcal{F}(A) = \phi$

Axioms are also used to codify statements about the nature of particular attributes. They facilitate additional storage of domain knowledge, by providing a well defined format in which to store information about an attribute.

## 2.5. *Attribute Hierarchies $A|_{F,\ V,\ B}$*

Attribute hierarchies occur naturally during the course of domain definitions. Starting with the base definition of attribute, 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 :

$\mathcal{V}(B)$ is a subset of $\mathcal{V}(A)$

$\mathcal{F}(B) \supset \{ f \mid_{\mathcal{V}(B)}$ such that $f \in \mathcal{F}(A) \}$

$\mathcal{X}(B) \supset (\mathcal{X}(A)$ restricted to $\mathcal{V}(B))$.

Let V be a subset of $\mathcal{V}(A)$, F be a subset of $\mathcal{F}(A)$, and X be a superset of $\mathcal{X}(A)$. Then $B = A \mid_{V,\ F,\ X}$ is the restriction B of A which has $\mathcal{V}(B) = V$, $\mathcal{F}(B) = F$, and $\mathcal{X}(B) = X$.

**Restriction Definition**

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:

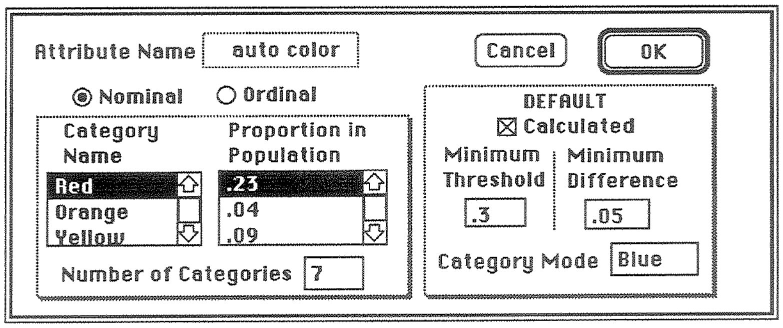
$\text{car\_color}|_{\text{warm}}$ and $\text{car\_color}|_{\text{cool}}$ where

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

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

## 2.6. *Population Parameters $\mathcal{PP}(A)$*

Associated with each attribute is a set of domain-specific population parameters, $\mathcal{PP}(A)$ that describe the probability distribution of values within the value set The characteristics of these distributions are used to organize class libraries, determine input checks for the runtime system, and determine default values for object properties.



Attribute Name: auto color

[✓ Nominal] Ordinal

| Category Name | Proportion in Population |
|---|---|
| Red | .23 |
| Orange | .04 |
| Yellow | .09 |

Number of Categories: 7

DEFAULT

☒ Calculated

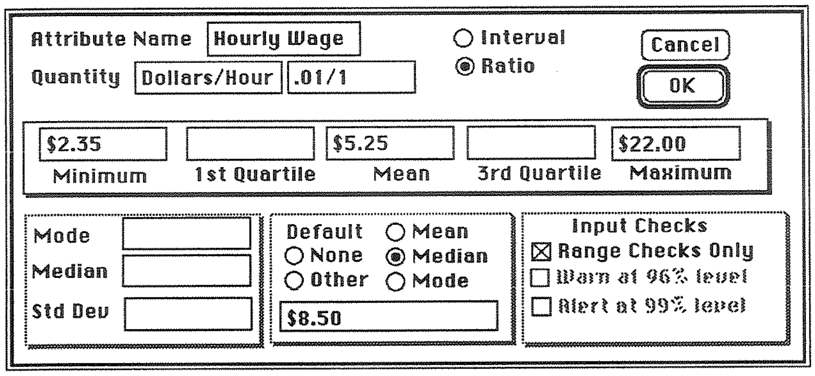
Minimum Threshold: .3

Minimum Difference: .05

Category Mode: Blue

Figure 8 — Population Parameters for Nominal & Ordinal Scales

Population Parameters vary with the scale type, and depending on the attribute being measured, certain parameters are more appropriate than others. For example, highly skewed ratio-scaled attributes such as salary might use the median instead of the mean to measure central tendency.


Attribute Name: Hourly Wage
Quantity: Dollars/Hour .01/1
○ Interval [✓ ◉ Ratio]

| | | | | |
|---|---|---|---|---|
| $2.35 | | $5.25 | | $22.00 |
| Minimum | 1st Quartile | Mean | 3rd Quartile | Maximum |

Mode
Median
Std Dev

Default: ○ Mean ○ None [✓ ◉ Median] ○ Other ○ Mode
$8.50

Input Checks
[✓ ☒ Range Checks Only]
☐ Warn at 96% level
☐ Alert at 99% level

**Figure 9 — Population Parameters for Interval & Ratio Scales**

## Organizing Class Libraries

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 10.

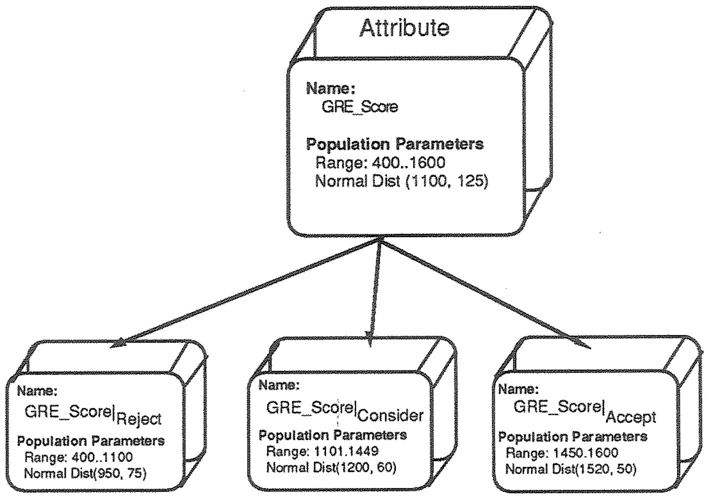


**Figure 10 — Attribute Hierarchies**

## Runtime Behavior

*Population Parameters* are available for the runtime system to use in checking user input and to determine defaults. Confidence interval checking is a more sophisticated form of ordinary range checking that matches runtime input values against the expected distribution of input. For example, an application designer might instruct the runtime system to suggest that the enduser be warned, with different levels of severity, about values outside of predefined confidence intervals.

## 2.7. *Initialization and Defaults*

Knowledge about the proper way to initialize an attribute is stored within a domain model as a function i, where i:Null --> V. Many attributes have clearly defined default values. For example, attributes that serve as counters such as number_of_times_checked_out usually begin at one or zero. Other attributes such as expected temperature depend on a particular domain.

Some default values can be automatically generated by the runtime system through the use of $\mathcal{PP}$(A). Consider the nominal scaled attribute of a person called main_transportation_method An application program for a transportation tax system might have the information defined with the following categories and proportions.

| main_transportation_method | |
|---|---|
| automobile | 61% |
| bus | 16% |
| bicycle | 12% |
| motorcycle | 3% |
| walk | 7% |
| other | 1% |

Consequently, the runtime system can generate automobile as the appropriate default. This style of default rule applies for all nominal scales across all domains (with tuning for the appropriate Threshold_value) and can be stated as follows:

For a set of categories $\mathcal{C}(m) = \{C_1, \ldots C_n\}$ in a nominal scale $\mathcal{N}(m)$,
If $\exists\, C_i$ (Proportion($C_i$) > *Threshold_value* ) then generate the default category_name( $C_i$)
where Proportion($C_i$) = $|C_i| / |\{C_1, \ldots C_n\}|$.

**Rule to determine default category for a nominal scale**

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].

## 3. Classes

Attributes are grouped into classes that provide templates for object creation. Classes consist of attributes, functions on more than one attribute, axioms, and special procedures for creation and deletion. The modeling technique formalizes classes as follows:

A class C consists of:

1. A set of attributes $\mathcal{A}(C)$.

   a. These attributes are divided into two disjoint groups:
   the fundamental,or primitive, attributes $\mathcal{P}(C)$ and the derived attributes $\mathcal{D}(C)$;
   that is, $\mathcal{A}(C) = \mathcal{P}(C) \cup \mathcal{D}(C)$ and $\mathcal{P}(C) \cap \mathcal{D}(C) = \phi$.

   b. The $\mathcal{A}(C)$. are also divided into two other disjoint groups:
   the key attributes $\mathcal{K}(C)$ and the nonkey attributes $\mathcal{N}(C)$;

2. A set of functions $\mathcal{G}(C) = \{g_i\}$, one for each $D_i$, such that $g_i$ is an *onto* function and $g_i : \mathcal{V}(P_0) \times \ldots \times \mathcal{V}(P_n) \times \mathcal{DOM}(S) \rightarrow \mathcal{V}(D_i)$, where : $\mathcal{P}(C) = \{P_0, P_1, \ldots P_n\}$, $\mathcal{D}(C) = \{D_0, D_1, \ldots D_m\}$, S is a 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.

3. A set of axioms or integrity constraints $\mathcal{X}(A)$.

4. An addition procedure, a, that uniquely associates an object of class C with each tuple of $\mathcal{V}(K_0) \times \ldots \times \mathcal{V}(K_l)$ $(\mathcal{K}_i \in \mathcal{K}(C))$.

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

**Class Definition**

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 system values such as current_date. The characterization of an attribute as primitive or derived is a function of the domain. For example, a rectangle with the attributes *length*, *width*, *perimeter*, and *area* can be viewed in a variety of ways. For most purposes, the attributes *length* and *width* would be considered to be 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 possible.

Age was introduced earlier in the discussion of units. If objects of the type person will persist in an application program for more than a year, age is derived from a person's birthdate and the current date. But this is a domain-specific decision. If one were creating a computer dating service, it might be better to store age rather than to derive it. The decision to derive, rather than store, age is based on the assumptions that 1) birthdate does not change, and 2) the value of the current date is a ubiquitous piece of information available to the system.

The definition of an attribute introduced the operations $\mathcal{F}(A)$. The class definition introduces operations, $\mathcal{G}(C)$, that compute the values of derived attributes. An example of such a function is cost_replacement for a library book.

```
cost_replacement(book) =
    weight(book)*shipping_cost_per_pound + cost(book) + overhead
```

Key attributes are the attributes whose values uniquely name an object at runtime. The selection of key attributes for an entity class is determined by the application domain and specified by the initialization function, i. This function stores the domain knowledge about initialization for that attribute.

The fourth and fifth part of the class definition refer to procedures that create and delete runtime objects. Creation procedures execute all of the initialization functions for the attributes of a class, and contain the information necessary to create uniquely identifiable objects. Deletion procedures maintain domain knowledge about unusual events such as the procedure to follow in the event of a lost or stolen library book.

## *3.1. Class Partitions*

Classes are partitioned into three types (domain-Independent, domain-Specific, and application Classes) and are organized into libraries by domain experts who use them to classify and store domain knowledge. Figure 11 illustrates some of the relationships among these kinds of classes. The application independent classes (vertical box on the left of figure 11) consist of entities, like books, that exist regardless of the existence of an application domain like libraries. The domain-specific classes (vertical box on the right of figure 11) represent abstract items such as lending objects that exist within domains. Application designers use the classes in the libraries as building blocks to construct program specifications. Although these distinctions depend in part upon the domain and the expert who creates the classes, they are useful classifications that allow the class libraries to be organized.

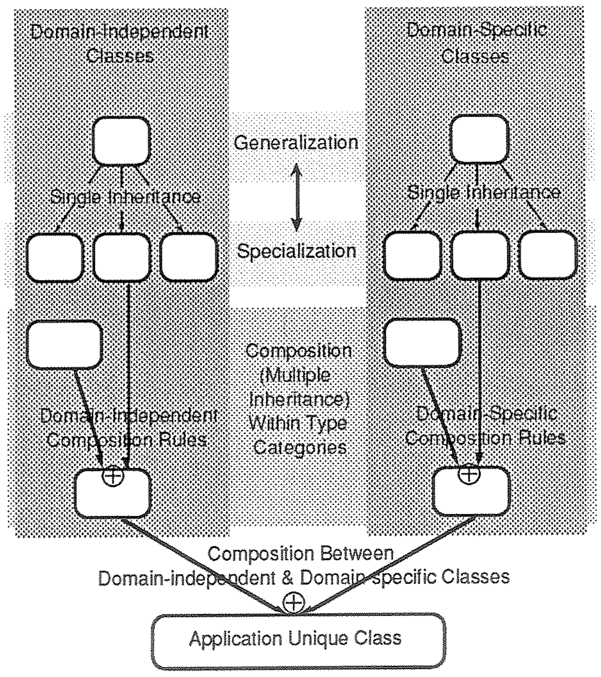

Domain-Independent Classes
Domain-Specific Classes
Generalization
Single Inheritance
Single Inheritance
Specialization
Composition (Multiple Inheritance) Within Type Categories
Domain-Independent Composition Rules
Domain-Specific Composition Rules
Composition Between Domain-independent & Domain-specific Classes
Application Unique Class

**Figure 11— Class Libraries**

## 3.2. *The Library Problem*

Figure 12 shows one view of the standard library problem. Explicit in the diagram is the idea that library books are composed from a domain-independent book class and a domain-specific lending object class. Similarly, library patrons are composed from a domain-independent person class and a domain-specific member class. Library books and patrons then participate in certain relationships that define the nature of a library. The diamond checkout application class is one such relationship.

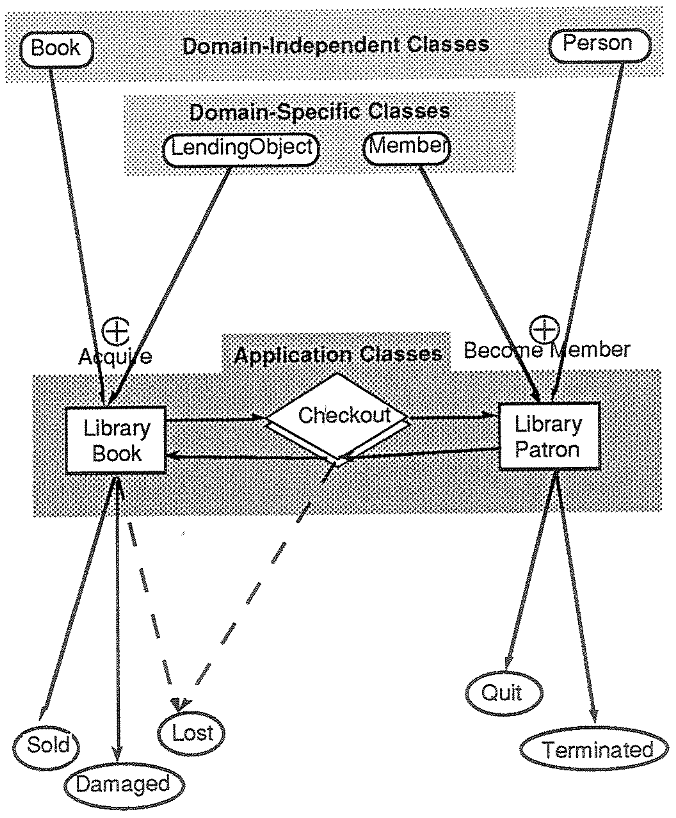


**Figure 12 - Library Objects and Operations**

## 3.3. *Domain-Independent Classes*

Certain classes exist regardless of the existence, or the lack of existence, of a particular application domain. Books and people, for example, exist both in and out of the domain of libraries. It is interesting to note that certain attributes of application independent entities, such as social security number for people and ISBN for books, have become so much a part of those entities that they are, for most purposes, primitive attributes.

Classes that exist independently of their appearance in an application domain will have properties that do not necessarily need to be mapped into the application, as well as attributes that change when they are composed with the attributes of classes in the application domain. A partial view of a book class as it exists without regard to the existence of a library is shown as follows:

```
Book = Class
        Title : Nominal_scale Title_Type
        Author : Nominal_scale Person_Name_Type
        ISBN : Nominal_scale:  ID_Number
        Copyright_date : Interval_scale date in years [x | x > 1800]
        Cost : Ratio_scale money in dollars
        Weight : Ratio_scale weight in pounds

OPERATIONS
      Age: Ratio_scale age in days (derived from Printing Date)
                .
```

**Domain-Independent Book Class**

## 3.4. *Domain Specific Classes*

There are many classes of objects such as order forms, picking tickets, checkout forms, library cards, card catalogs, and so on that exist because of an application. Figure 8 shows an *item_to_be_loaned* class that could be used for books, tapes, records, art, and other items that are to be loaned without charge.

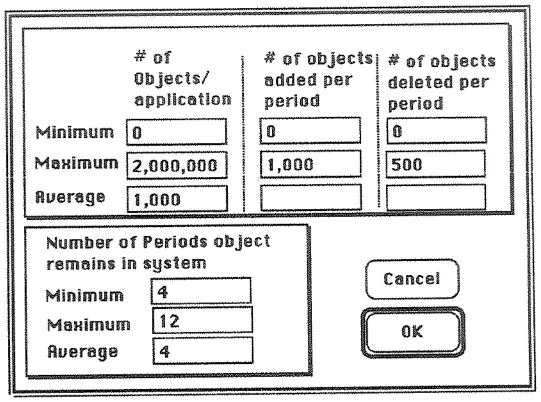
```
Item_to_be_Loaned = Class
      Unique_ID
      Loan_count : Ratio_scale count in integer
      Loan_status: Nominal_scale (loaned, not_loaned)
      Item_status: Nominal_scale (OK, missing, damaged)
      Acquisition_date : Interval_scale time in days
      Cum_Loan_time : Ratio_scale time in days
            .


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)
            .
            .
```

**Base Definition for an "Item to be Loaned"**

## 3.5. *Runtime Characteristics*

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. Figure 13 shows the capture of class population parameters that could be used to facilitate selection of the appropriate database algorithms for an application program.



| | # of Objects/ application | # of objects added per period | # of objects deleted per period |
|---|---|---|---|
| Minimum | 0 | 0 | 0 |
| Maximum | 2,000,000 | 1,000 | 500 |
| Average | 1,000 | | |

Number of Periods object remains in system

| | |
|---|---|
| Minimum | 4 |
| Maximum | 12 |
| Average | 4 |

**Figure 13— Class Population Parameters**

## *3.6. Encapsulation and Inheritance*

When encapsulated classes are used to create other classes through inheritance, breaks in the veil of encapsulation often occur. [Snyder 88] discusses the dynamic tension that often exists between encapsulation and inheritance. From a software engineering standpoint, OOP is good because it hides 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 this modeling technique, a balance is achieved between encapsulation and inheritance by reusable domain definitions that provide common building blocks for class construction as shown in Figure 2. In the figure, the box labeled *reusable domain definitions* refers to a library of scales, units, and attributes that stores information about domain properties. Class hierarchies establish the structure of a domain in terms of those properties. Common definitions allow a balance between encapsulation and inheritance by hiding attribute implementation details, while facilitating the construction of large libraries of classes.

## 4. Composition Within Class Libraries

Composing the properties of two classes into a single class with its own set of properties is not a problem that can be solved by simply taking the union of the attributes of the two classes from which the resultant class is derived. Our view of composition combines views of the artificial intelligence community [Brachman 85] with more standard object-oriented views. In our modeling technique, it may be necessary to introduce new attributes as well as to rely on domain knowledge to resolve conflicts among existing ones.

### *4.1. Creating New Attributes*

Application Classes are composed from domain-independent and domain-specific classes, as in the case where Book and Item_to_be_Loaned are composed to produce the class Library_Book, as shown below. Application classes in a domain model must include a creation procedure that not only triggers the initialization functions for all attributes in the class, but also creates a unique name for an object. Constructing application classes sometimes requires that additional Key attributes be created so that the objects created from those classes will be uniquely identified. For example, people usually identify a single book by its title, or perhaps by its author and a portion of its title. Bookstores identify books by their titles, or their ISBN identifier. Libraries also identify books by their titles (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).

```
Library_Book  = Class
    Unique_ID:
        ISBN
        Copy_Number
    Title : Nominal_scale Title_Type
     Author : Nominal_scale Person_Name_Type
     Copyright_date : Interval_scale date in years [x | x > 1800]
     Cost : Ratio_scale money in dollars
    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 from Acquisition_date)
        .
        .
```

**Library_Book = Book ⊕ Item_to_be_Loaned**

The result of this composition is shown. Obviously, the Loan_count, and the Cumulative_loan_time should be initialized as zero. The Acquisition_date will be used to determine the length of time that the book has been in the library (its relative age). Unique_ID is a slot that tells the system that an Item_to_be_loaned must have a unique identifier.

## *4.2. Conflict Problems*

[Snyder 88] describes the current three approaches to solving conflict problems in multiple inheritance. These solutions are syntactic, and while appropriate from the standpoint of object-oriented programming languages, they fail to address domain issues. The example in figure 14 (a modification of the automobile example from [Cardelli 85]) illustrates the problems which arise in multiple inheritance.

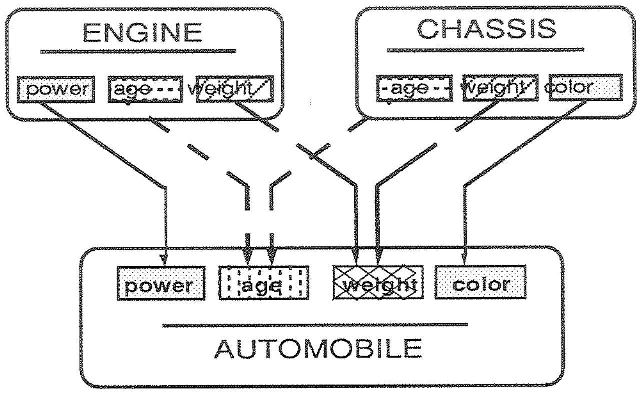


**Figure 14-- Determining Age & Weight of an Automobile**

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 following ways:

```
Average_component_age
    automobile.age := (engine.age + chassis.age)/2
Sum_of_component_ages
    automobile.age := engine.age + chassis.age
Age_of_oldest_component
    automobile.age := max(engine.age, chassis.age)
Ages_of_components
    automobile.age :=set of {engine.age, chassis.age}
Age_since_manufacture
    automobile.age := current_date - Automobile.create_date
```

Within the domain of car buyers, most people would say that automobile age would best be calculated by the *Age_since_manufacture* operation based on the date that items are assembled to form the basis for the age. Of course, other domains will have other rules, 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 traveled.
- Age of automobile engines is measured in years of service.
- Age of boat and airplane engines is measured in operations hours.

The semantics of composition are partially determined by the domain knowledge stored by the attribute. For example, 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. In specific terms such a rule might read, "*When you combine two classes, each of which have a weight attribute, into a composite class, 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.*"

The rule that allowed us to determine how to combine the weight attributes is:

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

1. $A_1$, $A_2$, and $A_3$ all must have the same scale measure $\mathcal{M}(A)$.
2. The measurement granularity of the unit of $A_3$ must be at least as coarse as the most coarse of the measurement granularity of the units of $A_1$ and $A_2$.
3. The semantics of the composition are determined by the attribute.

**Rule for Composition of Interval & Ratio Scaled Attributes**

Composing classes requires a large component of domain knowledge that includes the semantics of attributes and their combination. For example, it makes sense to add two weights together, but it makes very little sense to add two ages or temperatures, even though they are both ratio scales. In short, the semantics captured by the representation of attributes in terms of scales and granularity aids in conflict resolution, although there will be cases in which additional more explicit domain knowledge is still required.

## 5. Research Context

Our research can be viewed as automated programming, a specialized area which can be approached from the many different perspectives which are discussed in [Barstow 84], [Balzer 85], and [Rich 88]. We have approached the research from the standpoint of capturing knowledge about a domain, a pursuit that is applicable to aspects of software engineering that extend well beyond simple program generation.

Domain knowledge is not usually specified explicitly . Although it is an integral part of program specifications, it often appears only between the lines of specification documents. And while domain knowledge is an essential part of fourth generation and other special purpose languages, it is almost never formally represented and rarely codified in a way that allows it to be easily used or reused by those unfamiliar with a domain. 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],.
- is poorly defined and frequently takes years to gather [Curtis 88].

Research on application domain modeling ([Arango 89], [Greenspan 86], [Lubars 88], [Reubenstein 89], [Robinson 89]) [7] is driven by a variety of operational goals that include domain analysis, requirements elicitation, reverse engineering, training, capturing and resolving design decisions, as well as system development and evolution.

## 6. Conclusion

This paper has described a formalized domain model and modeling technique which we designed to represent relevant portions of the domain knowledge required to specify and implement application generators. The focus was on a representation technique to support the meta-level aspects of such a system—a generator for narrow domain application program generators.

Parameterized attributes are used to construct classes that capture knowledge about application domains. Domain-independent and domain-specific classes can then be composed into application classes, such as library books, that in turn can be instantiated within an application program. The automobile example showed the role of scales as constraints, and illustrated how composition can be aided by domain knowledge.

A system prototype, Ozym, has been developed to experiment with domain model generation. This system is being used to iteratively refine the modeling technique and consequent domain models.

### *Acknowledgements*

Thanks to Gerry Williams for his comments in reviewing this paper.

## References

[Adelson 85] Adelson, B., Soloway, E., "The role of domain experience in software design, in *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, Nov. 1985.

[Arango 89] Arango, Guillermo, Domain Analysis: From Art Form to Engineering Discipline, *Proceedings of the 5th International Workshop on Software Specification and Design*, Pittsburgh, PA, May 19, 1989, pp 152-159.

[Balzer 85] Balzer, R., "A 15 year perspective on automatic programming, in *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, Nov. 1985.

[Barstow 84] Barstow, D., "A perspective on automatic programming," AI Magazine, 5:1, pp. 5-27, Spring 1984.

[Booch 86] Booch, Grady, "Object-Oriented Development," *IEEE Transactions on Software Engineering*, Feb 1986, pp. 211-221.

[Brachman 85] Brachman, R, "I Lied about the Trees',Or, Defaults and Definitions in Knowledge Representation," *The AI Magazine*, Fall 1985, pp 80-92.

[Cardelli 85] Cardelli, L. and Wegner, P.; "On Understanding Types, Data Abstraction, and Polymorphism"; *Computing Surveys*; 17, 4, Dec 85; pp 471-522.

[Coombs 53] Clyde H. "Theory and Methods of Social Measurement", in Leon Festinger and Daniel Katz (ed.), *Research Methods in the Behavioral Sciences,* New York: Holt Rinehart, and Winston, 1953, pp. 471-535

[Coombs 60] Coombs, C.H, "A Theory of Data" *Psychological Review, Vol 67,* 1960, pp. 143-159.

[Cox 86] Cox, Brad, *Object Oriented Programming,* Addison-Wesley, 1986.

[Curtis 85] Curtis, Bill, "Measurement and Experimentation in Software Engineering", *Proceedings of the IEEE,* 1980, 68(9), pp. 1144-1157.

[Curtis 88] Curtis, B. Krasner, H, Iscoe, N A Field Study of the Software Design Process for Large Systems, Communications of the ACM, 31, November 1988, pp. 1268-1287.

[7] This list represents only some of the current domain modeling projects.

[Dahl 66] Dahl, O.J. and Nygaard, K.; "SIMULA—an Algol Based Simulation Language"; *Communications of the ACM* 9; 1966; pp. 671-678.

[Danforth 88] Danforth, Scott, Tomlinson, Chris, "Type Theories and Object-Oriented Programming, *ACM Computing Surveys*, Vol. 20, No. 1, March 1988.

[Goldberg 83] Goldberg, A., and Robson, D.; *Smalltalk-80: The Language and its Implementation"*; Addison Wesley, Reading, Massachusetts; 1983.

[Greenspan 86] Greenspan, S., Borgida, A., Mylopoulos, J., "A Requirements Modeling Language and its Logic," *Information Systems*, V11#1, Pergammon Press, 1986, pp. 9-23.

[Guttag 77] Guttag, John, "Abstract Data Types and the Development of Data Structures, *Communications of the ACM*, June 1977, pp 396-404.

[Iscoe 88] Iscoe, Neil, Domain-Specific Reuse: An Object-Oriented and Knowledge-Based Approach, Software Reuse: Emerging Technology; editor Will Tracz, IEEE Computer Society, Washington, DC, September 1988, pp. 299-308.

[Iscoe 89] Iscoe, N, An Object-Oriented and Knowledge-Based Approach to Domain-Specific Programing, Doctoral Dissertation, University of Texas, Austin, Texas, 1989 (in process).

[Jackson 83] Jackson, M, System Development, Prentice-Hall, 1983.

[Keeney 76] Keeney, R, and Raiffa, H, *Decisions with Multiple Objectives: Preference*s and Value Tradeoffs, John Wiley and Sons, 1976.

[Kemmerer 85] Kemmerer, R., "Testing Formal Specifications to Detect Design Errors, in *IEEE Transactions on Software Engineering*, Vol. SE-11(1):32-43, January, 1985.

[Liskov 74] Liskov, B, and Zilles, S, "Programming with Abstact Data Types, " Computation Structures Group, Memo # 99, Project MAC, MIT, Cambridge, Mass, 1974.

[Lubars 88] Lubars, Mitch, Domain Modeling Representation, MCC Technical Report, STP 366-88, November 1988.

[Meyer 88] Meyer, Bertrand, *Object-Oriented Software Construction*, Prentice Hall, New Jersey, 1988.

[Rich 88] Rich, Charles, "Automatic Programming: Myths and Prospects"; *Computer*, August 1988, pp. 40-51.

[Reubenstein 89] Reubenstein, H, Waters, C, "The Requirments Apprentice: An Initial Scenario", *Proceedings of the 5th International Workshop on Software Specification and Design*, Pittsburgh, PA, May 19, 1989, pp 211-218

[Robinson 89] Robinson, Bill and Fickas, Stephen, "Integrating Multiple Specifications Using Domain Goals," *Proceedings of the 5th International Workshop on Software Specification and Design*, Pittsburgh, PA, May 19, 1989, pp. 219-226

[Shortley 73] Shortley, George and Williams, Dudley, *Elements of Physics*, Prentice Hall, 1973.

[Snyder 88] Snyder, Alan; "Inheritance and the Development of Encapsulated Software Systems" in *Research Directions in Object-Oriented Programming*, ed. Shriver & Wegner, MIT Press, 1988, pp. 165-188.

[Stefik 85] Stefik, & Bobrow, G. Object-Oriented Programming: Themes and Variations, *The AI Magazine*, Winter 1985, pp. 40-61.

[Stevens 46] Stevens, S.S. On the Theory of Scales of Measurement, *Science* 103: 1946, pp. 677-680.

[Torgerson 58] Torgerson, Warren, Theory and Methods of Scaling, John Wiley & Sons, 1958.

[Wegner 88] Wegner, P, "The Object-Oriented Classification Paradigm", Research Directions in Object-Oriented Programming, ed. Shriver & Wegner, MIT Press, 1988, pp. 479-560.

[Wing 88] Wing, J., "Study of Twelve Specifications of the Library Problem," Technical Report, Department of Computer Sciences, Carnegie Mellon University, 1988.