Functional Lists: Object-Oriented Design Classes for MIS Applications

William R. Bitman

System Automation Corporation
8555 Sixteenth St., Silver Spring, MD 20910

ABSTRACT

An important process of object-oriented engineering is the identification and definition of objects and classes. Operationally, there are three kinds of objects: fundamental, director, and design. Classes define traits of objects. For Management Information Systems (MIS), fundamental classes are abstractions of stored and displayed data referenced in the requirements. Director objects serve as the glue that combines objects to accomplish the requirements. Design classes are defined by analysts to further modularize the application, thereby reducing module complexity. This paper formulates the functional list design class. A functional list contains rules that determine whether an item belongs on a list. *Fundamental* list classes, in contrast, provide the mechanical manipulation operations of lists and list items. In many MIS systems, functional lists help organize code into reusable operations of low complexity. Functional lists are instantiations and often implemented as derived types, thus utilizing inheritance features.

Two objectives of software engineering are reliable programs and high productivity. One way to increase reliability is to decrease software complexity of the individual software units of the system [8, 14]. A software unit is a discrete collection of executable statements called a module [9]. Low module complexity makes testing easier because a complete test suite consisting of a manageable number of test cases can be defined based on the limited number of unique paths and critical data values [10]. Reliability of the system as a whole increases when each module is thoroughly tested. Productivity can be increased through reuse of software units because the total amount of code is reduced [1]. Thus, reuse and low complexity are means to attaining software engineering goals.

Low module complexity can be achieved by dividing a system into a greater number of modules. This distributes the inherent complexity, thereby decreasing the average module complexity [7, 12]. There are many methods for dividing a system into more parts. Among these, object-oriented methods have the added benefit of facilitating identification of common algorithms, which increases reuse. Even

greater reuse can be achieved by using code-reducing abstraction facilities of Ada, such as genericity [6, 11] and derived types [13].

By combining object engineering with powerful Ada language features, object-oriented Ada methods enable significant increase both in reliability and productivity by decreasing module complexity and total size. Figure 1 displays these relationships.

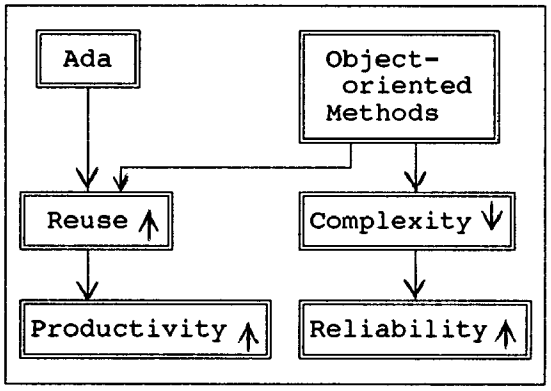

Figure 1: Relationships of factors for achieving engineering goals.

In Ada, a module is a subprogram. Ada modules exist as stand-alone subprograms or as collections of subprograms in packages. A package of subprograms may also export a type; if it does, it is an abstraction of a class.

CLASSES

A class defines characteristics that many objects share [4]. An object has a value (or state) and a set of relevant operations [3]. The Ada package is a powerful implementation of a class. The exported type defines the allowed values for objects of that class, while exported operations provide relevant capabilities. Modules that need an object, "with" the package that defines its type, and then can declare variables (i.e. objects) of the exported type and use its exported operations. There are three major categories of classes: (a) fundamental, (b) director, and (c) design. Most fundamental and director classes are abstractions of the problem space and are derived directly from requirements, whereas design classes are created by analysts to provide solution space items [15].

Fundamental classes define the basic data types in the system. In Management Information Systems (MIS), most fundamental classes are abstractions of data stored on disk and data displayed to the user. These can be simple, composite, or structural. The data type of a simple class does not have discernable parts, whereas that of a composite class does. Structural classes are lists, stacks, queues, etc.

Director objects perform processing that is readily understandable in terms of end-user's objectives. Executable code in directors is often referred to as glue that binds modules to accomplish functional objectives. This glue must be implemented as large complex modules unless there are supporting classes that distribute the complexity. Design classes provide this support.

DESIGN CLASSES

Design classes are generally not directly evident from requirements. Most arise during the design phase as products of problem-solving activities, and,

therefore, are purely solution space classes. They are abstractions of intermediate objects that handle data processing and perform algorithms. In the process of creating design classes to solve requirements, new fundamental classes also may be created.

FUNCTIONAL LIST CLASS

The functional list is proposed in this paper as a design class category which is advantageous for many MIS applications. A functional list class is a package which (a) exports a list type which is specialized but whose element is a basic type (e.g. college classroom location list type is exported by a functional list class whose element on the list is a location, which is a fundamental type used by many objects throughout the system); and (b) exports operations that give the list its characteristics in the application (e.g. build a list of course locations where this course, which needs special equipment, can be scheduled).

Functional lists are advantageous for applications in which many lists have the same data type but are functionally distinct and are utilized simultaneously. Designers can insure that functionally distinct lists are not accidently used for unintended purposes. The use of strong typing to create a functional list identity utilizes the Ada compiler as a validity checker.

Another benefit of functional lists is that executable software becomes highly organized. An alternative to having many functional list classes which preserves this benefit is to group all operations for an element type (e.g. Location) into one list package (e.g. Location_List). This implementation maintains the advantage of organizing code into small operations. The major drawback is that compiler type checking is no longer used to check functional integrity. This problem will be described under functional operations package in the next section. A practical problem is that the number of operations that belong to such a general list package grows at a very fast rate because so many different requirements relate to it. The specification changes often, and when it does, an enormous number of units must be recompiled.

The steps in the process of identifying functional list classes are:

1. Determine a set of data lists with which requirements can be satisfied.

2. For each list, identify relevant operations.

3. For each list that has an element of a composite nature, determine the set of fields (e.g. record components) that uniquely identifies an item on the list (i.e. key).

4. Determine whether there is unnecessary duplication of classes. A class attains identity through the uniqueness of its key and operations. If there are operations relevant only to a subset of objects, multiple list classes with identical keys are justified.

5. Describe each operation of each list class. The description, which can take the form of prose, pseudocode, or code, gives a clear idea of the complexity of the module and identifies any additional classes needed. An overly complex module may be simplified by defining new operations within existing classes or by defining new classes. A cyclomatic

complexity of 10 is considered the reasonable limit for a module [10]. With this method, complexity is monitored early in the life-cycle so that it does not go unnoticed and become difficult to manage during or after implementation.

The above steps are performed iteratively until a complete, correct, and consistent system is designed.

IMPLEMENTATION

The implementation of functional list classes can take many forms. These are presented here in order of increasing degree of abstraction and encapsulation. The fundamental class and generic fundamental list class first presented are referenced throughout the examples.

```
package Thing is
  type THING is (X, Y, Z) ;
  procedure A
    ( This_Thing
        : in     THING ;
      Some_Data
        :    out INTEGER ) ;
  function B
    ( Something
        : in INTEGER )
   return THING ;
end Thing ;
```

Listing 1: Fundamental Class

```
generic
  type ELEMENT is private ;
package List_Structure is
  type LIST is private ;
  procedure Add
    ( This
        : in     ELEMENT ;
      To
        : in out LIST ) ;
    procedure Delete ... ;
    procedure Update ... ;
    function Item ... ;
    procedure Clear
      ( This
          : in out LIST ) ;
    function Length
      ( Of_List
          : in LIST )
     return NATURAL ;
private
  type NODE ;
  type LIST is access NODE ;
end List_Structure ;
```

Listing 2: Fundamental List Class

1. Basic List

A basic list is an instantiation of a list structure with the desired data element type. It serves as the starting point for functional lists. List and element manipulation functions are inherited from the generic package, but it has no operations additional to the fundamental class.

```
with Thing ;
with List_Structure ;
package Things is new
  List_Structure
    ( ELEMENT
        => Thing.THING ) ;
```

Listing 3: Basic List Class

2. Functional Operations Package

A functional operations package exports operations that are relevant to a discernable category of lists. For example, a package can be defined as being concerned with operations for lists of office locations. Although it does not export a type, member operations are limited to those that deal with some aspect of the list category for which the package was created. However, list objects and parameters in this package are typed as the basic list type (e.g. Location_List). This implementation has the advantage of organizing code functionally around a "conceptual" class, although the package is not a true class because it does not export a type.

In some situations this implementation is sufficient and acceptable. The disadvantage is that the capability of the compiler to act as an integrity checker is lost. For example, there is a functional operations package which includes an operation to return the business hours for each insurance agency office location on a list. A calling module can send a list of parking meter locations to this operation because both the list of office locations and the list of parking meter locations are of type Location_List.LIST and the compiler cannot detect the context mismatch.

```
with Thing ;
with Things ;
package Functions_For_Things is
  function Build
    ( Something
        : in INTEGER )
   return Things.LIST ;
  procedure Sort
    ( List
        : in out Things.LIST);
  procedure Find
    ( This
        : in     Thing.THING ;
      In_List
        : in     Things.LIST ;
      Position
        :    out INTEGER ;
      Count
        :    out NATURAL ) ;
end Functions_For_Things ;
```

Listing 4: Functional Operations Package.

3. Visible Functional List

A visible functional list class is a package that (a) exports a visible type derived from the type exported by a basic list class, and (b) exports its own unique relevant operations. The level of abstraction attained by this implementation is the minimum necessary to have a true functional list class. There may be many functional lists derived from the same original basic list. For example, there are many reasons to build and use lists of dates. Each identifiably different reason should be implemented as a separate class, each with its specific exported list type.

```
with Thing ;
with Things ;
package Functional_Things is
  -- This is a derived type:
  type FUNCTIONAL_THINGS is new
    Things.LIST ;

  function Build
    ( Something
        : in INTEGER )
   return FUNCTIONAL_THINGS ;

  procedure Sort ... ;

  procedure Find ... ;

end Functional_Things ;
```

Listing 5: Visible List

For example, in a college student information system, there are many date lists needed. Dates is a class which is an instantiation of a generic list with element Date.DATE. Operations needed for course dates of one-day seminars of various titles are in one package. There also are lists of dates of football games. These operations are provided in a different package. Class package Seminar_Dates exports SEMINAR_DATES list type. Objects of this type have items of type Date.DATE on it because Seminar_Dates is derived from Dates.LIST (see the derived type in Listing 5). The class exports operations including Build which builds a list of SEMINAR_DATES on which an input seminar of a specific course title is scheduled. Class Football_Dates exports its own type and has a Build_Home function which returns a list of home game dates for a given opposing team. Although both lists have the same element (i.e. Date.DATE), they are functionally distinct and recognized as such by the compiler. Therefore, a seminar date list object cannot be submitted to the football date list operations. However, dates on a seminar date list can be directly compared to dates in a football date list because they are objects of the same type.

4. Private Functional List

A private functional list class is similar to a visible functional list class except that the list type is private, and, therefore, all fundamental list operations that the designer desires the caller to have access to, must be explicitly exported. A private functional list class is useful when there are specific fundamental list operations that must be prohibited from the caller.

```
with Thing ;
with Things ;
package Functional_Things is
  type FUNCTIONAL_THINGS
    is private ;

  function Build
    ( Something
        : in Build )
   return FUNCTIONAL_THINGS ;

  procedure Sort ... ;
  procedure Find ... ;

  --  The following operations
  -- must be explicitly defined
  -- because the type is
  -- private.
  procedure Add ... ;
  procedure Delete ... ;
  procedure Update ... ;
  function Item ;
  procedure Clear ... ;
  procedure Length ... ;

private
  type FUNCTIONAL_THINGS is new
    Things.LIST ;
end Functional_Things ;
```

Listing 6: Private List Class

5. Specialized Related Lists

A specialized type class exports a list whose element type itself (not just the list type) is derived from a fundamental type. Elements of two different functional list classes derived from the same basic type cannot be directly compared. Explicit conversion is needed for comparisons of items on objects of different list classes. Therefore, this implementation is most appropriate when there is an inherent difference between the two element types.

For example, there is a class Pop_Center (population center) that exports type POP_CENTER. Class City which exports type CITY is derived from Pop_Center.POP_CENTER and adds its own operations. Package Cities is an instantiation of a generic list package with element City.CITY. Similarly, package Town exports derived type TOWN and various operations. Towns is the functional list package. Items on a Towns.TOWNS list and items on a City.CITY list are different types, and, therefore, must be processed through explicit type conversions. This is in contrast to the previous implementation approaches in which, although the list types are different types, the elements on the lists are the same type.

6. Private Specialized Lists

The previous implementation level can be extended to private type elements. The reason for selecting private over visible depends on the degree of selectivity of fundamental operations. With a private implementation, elements on lists must be process through their own Construct and Value operations before comparison and assignment operations can be performed between items of different types.

METRICS

There are two metrics of interest: complexity and reusability.

Cyclomatic complexity is an indication of software decisional logic. It is easy to calculate for each module, and therefore, can be measured by every coder and checked by every project leader early

in development. Cyclomatic complexity is directly related to the number of test cases needed to achieve acceptable reliability. Keeping complexity low increases the chance of finding errors early in the development life-cycle. The further along in the life-cycle an error is detected, the costlier it is to correct [2]. Complexity on the module level can be extended to functional groups so that total complexity can be easily measured also [10]. This insures reliability tracking as integrated testing incrementally proceeds.

The reusability of software is the potential of a module to be reused and is not an immediately calculable metric. At the design stage, the level of expected reuse can be estimated based on knowledge of the application. The full extent of reusability may be greater due to usefulness to future projects. Reuse, on the other hand, is the actual number of times a module is used and is a more concrete metric to deal with. However, a module may be highly reusable, but may in fact, never be used for a particular project. During design, there are guidelines that increase likely eventual reuse. On a basic level, a package must export most operations if reuse is to become a reality. Only sensitive capabilities should be kept local. A pragmatic approach to quantify reuse is to sum the number of additional times modules are called in an application.

Functional lists were successfully applied to a project for the Army in which existing code for a training assignment system was redesigned and implemented. The original application was designed using structural decomposition in a conventional procedural (non-Ada) language. Metrics of the design classes of a discrete segment of the original implementation of the application were calculated and are summarized in Table I. Each module is given a name code. The number of executable statements, number of nodes as defined by McCabe and Butler [10], and the cyclomatic complexity are shown. The requirements have a cyclomatic complexity of 102, as noted by the total complexity of the conventional approach. The modules of the original implementation have an average complexity of 26, well above the acceptable limit.

Module	Executble Statments	Nodes	Complx
F1	105	53	31
F2	68	27	18
F3	60	34	22
F4	125	54	31
Sum	358	168	102
Avg	90	42	26

Table I: Metrics of non-Ada code.

Metrics for the object-oriented Ada implementation of this system are shown in Table II. Seven classes were identified by using a functional list design. Complexity was divided into 18 modules, with an average complexity of 4. Although the Ada modules perform the same functionality as the former system, they have a total complexity of only 71. Note that inherited operations of generic instantiations (e.g. length, clear, item, search, sort) were also used but are not included because they form a library of dependable, tested modules. Only design classes are included.

Module	Executbl Statmnts	Nodes	Complx
T1_B1	15	8	4
T1_B2	18	5	3
T1_R1	5	4	2
T1_R2	7	7	5
T2_B1	24	4	3
T2_B2	22	13	2
T2_C	37	17	8
D1_B	20	11	5
D1_G	9	8	4
D2_B1	14	4	2
D2_B2	7	2	2
L1_B1	21	4	2
L1_B2	8	2	2
L2_B1	7	9	7
L2_B2	26	22	9
L2_B3	14	12	7
L3_B1	14	4	2
L3_B2	12	2	2
Sum	280	138	71
Avg	16	8	4

Table II: Metrics for Ada implementation.

The module name code scheme is based on class category and operations. The first character signifies the class category. There are two functional training list classes, T1 and T2; two functional date list classes D1 and D2; and three functional location list classes L1, L2, and L3. The fourth character position in the name signifies the kind of operation of the module. B is a list build. Note that many classes have multiple build operations, one for each set of qualifying rules used for different purposes. R is an operation that removes items from a list based on certain criteria. G is a get range. Operation C is a check operation in which the validity of data after a passage of time is double-checked.

Table III compares the metrics of the two implementations. The Ada system has almost 5 times as many modules, but the total complexity is 30% less. This is partly due to the reuse achieved by the fact that four of the modules are called twice, and one is called three times in this particular application. Unit testing was made easier and more concrete due to the fact that the average module complexity is a manageable value of 4.

	Old System	Ada
Number of Modules	4	18
Total Complexity	102	71
Average Module Complexity	26	4
# modules called 2x	0	4
# modules called 3x	0	1
Reuse Index	0	6

Table III: Comparison of metrics for Ada system and the original non-Ada system.

In the application, the reusable modules provided three independent paths that the director software needed. Integration testing was performed in an

orderly fashion based on the body dependencies. These are shown for each of the three calling paths in Figures 2a, b, and c. An arrow to a module is a call to that module. Modules that have no dependencies were tested first. They tend to be placed along the outside edge of the diagram. The next layer in was tested next, etc. Many modules were called multiple times within these calling paths. The additional calls are indicated by asterisks. The accrued benefit of reuse is strikingly evident by the fact that operation T2_C in Figure 2c depends completely on reused operations. So although testing T2_C is based on its cyclomatic complexity of 8, it has an effective functional complexity (design complexity [10]) of 23. Testing is manageable because all the other modules have already been tested.

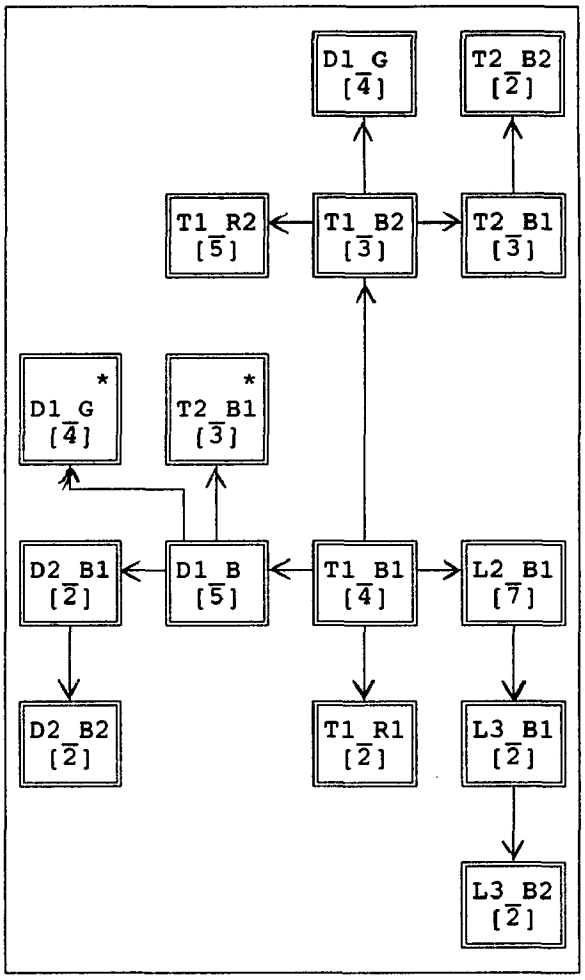

Figure 2a: Ada module execution dependencies for the major functional area in the Army application. Complexity value is in brackets.

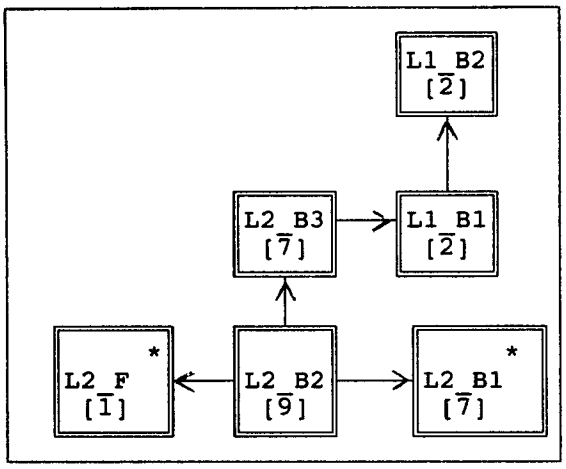

Figure 2b: Ada module execution dependencies for the second functional area in the Army application. Complexity value is in brackets.

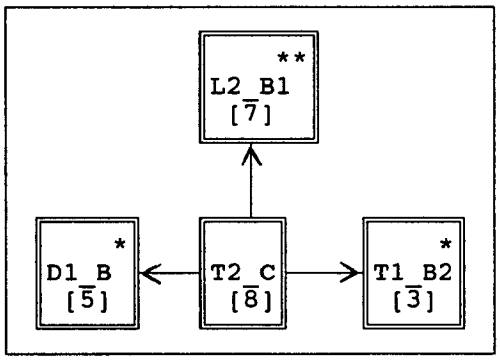

Figure 2c: Ada module execution dependencies for the third functional area in the Army application. Complexity value is in brackets.

EXAMPLE

The following example is patterned after the military training assignment system, to which functional lists were applied. Many of the complex aspects of the original application are conveyed to some degree in the example.

Methodology Steps

The following are the basic steps to develop a program that uses objects to accomplish requirements:

1. Determine the complete and consistent functional requirements.

2. Code the fundamental class specs and compile.

3. Code the director object.

4. Identify design classes and code specs.

5. Describe bodies of operations of fundamental and design classes.

6. Add new fundamental and design classes and/or new operations to existing classes as needed.

7. Code bodies, compile and test as needed operations become available. Incremental integration testing occurs in this step.

8. Repeat steps 4, 5, 6, & 7 until an acceptably low complexity level is achieved for each module.

9. Link the application and test.

Requirements

The following hypothetical specification serves as an example on which the principles of functional list design class development will be applied and illustrated.

Machiners Training Corp. (MTC) offers courses in heavy machinery operation and maintenance to employees of other companies. There are 2 categories of training: basic (BT) and special (ST). Each course lasts 1 week from Monday to Friday. BT is given in 2 forms: heavy and light. Those people who

cannot perform heavy lifting must take the light training. There are 6 different categories of ST, two of which require heavy lifting. A person who wants to train in a heavy-lifting ST must have taken heavy-lifting BT.

BT is a prerequisite for any ST. There are 3 employee situations: (a) those new to the job who need only BT, (b) those new to the job and need BT and an ST, and (c) those who already have taken BT, but now need ST. When both are taken, it is preferable to have them taken as close in time as possible. For a particular employee, a company may specify a prioritized list of specialties that are acceptable. The company wants the employee trained in the specialty of highest priority, but does not want training delayed more than four weeks if training in a lower priority specialty is available sooner.

MTC has 5 BT facilities throughout the country. There is only one BT class taught per location at any time and each has a maximum seating capacity. Three of them are fully equipped and can support heavy or light training (but only one category at a time), whereas 2 can support only light training. Four of the ST facilities are located at a BT location. Some BT seats are set aside for people taking an ST on the following week at the same site (privileged seats). This is the preferred arrangement because it reduces travel costs for the client company. At a certain number of days before a BT class starts, unassigned privileged seats lose their special status, in order to minimize wasted seats. The threshold number of days before the BT start date at which time this occurs is different for each BT location.

In general, travel costs are to be minimized. MTC updates a list of location pairs. For each major city in the U.S., there is another city which is least expensive to travel between. Likewise there is a city that is second lowest in cost, etc.

There are 8 ST locations. Many have multiple classrooms. Each classroom is equipped for a specific course. However, some courses need only a subset of the equipment of another course. Each class has a maximum seating capacity.

Companies find that when employees have been trained in homogeneous groups, they have difficulty dealing with others. Therefore, MTC tries to schedule both men and women for all classes. Also, in ST classes for which the course does not require heavy lifting, MTC tries to have both those who took heavy-lifting BT and light-lifting BT.

MTC reserves seats on an as-come basis. It does not save up a number of training requests to batch-optimize reservations. Usually, a client company representative requests the reservation for an employee and has the earliest start date and a list of any dates the employee cannot attend. The user enters the data and receives a list of training possibilities and selects one.

Design Strategy

To start the design process, we may either write the director object or we can define the fundamental objects that are apparent from the requirements. These two steps are independent and can be performed in parallel, if so desired. For

this example, the an overview of the major fundamental classes will be specified first.

Fundamental Classes

From the requirements, there are many fundamental classes readily identifiable. In an existing development environment, many of these would already be in reusable libraries. In order to form a basis for the example, most of these classes are sketched below. For each type definition below, there is a package that has the same name as the type. For many of these classes, there are a number of operations that can be identified from the requirements. Some are specified here.

```
type BT_CLASS is record
  Start
    : Date.DATE ;
  Place
    : Location.LOCATION ;
  Lifting_Required
    : Weight.WEIGHT ;
  Number_Of_Privilege_Seats
    : NATURAL ;
  Regular_Seats_Taken
    : NATURAL ;
  Privilege_Seats_Taken
    : NATURAL ;
end record ;
```

```
type BT_CLASSROOM is record
  Place
    : Location.LOCATION ;
  Capacity
    : NATURAL ;
  Highest_Weight
    : Weight.WEIGHT ;
  Privilege_Threshold
  --  Number of days before
  -- which, privilege
  -- seats are no longer
  -- special.
    : NATURAL ;
end record ;
```

```
type DATE is record
  The_Month
    : Month.MONTH ;
  The_Day
    : Day.Day ;
  The_Year
    : Year.YEAR ;
end record ;

function Day_Name
--  Return the day name
-- of a date.
  ( Of_This_Date
      : in DATE )
 return
   Day_Of_Week.DAY_OF_WEEK;

function Next_Monday
--  Return the date of the
-- Monday after the entered
-- date. If the entered date
-- is Monday, it is returned.
  ( On_Or_After_This
      : in DATE )
 return DATE ;
```

```
function Previous_Monday
  ( On_Or_Before_This
      : DATE )
 return DATE ;

function Difference
--   Calculate the number
-- of days between 2 dates.
-- If first date is later than
-- first, negative value
-- results.
  ( Date_1
      : DATE ;
    Date_2
      : DATE )
 return INTEGER ;
```

```
type DAY_OF_WEEK is
  ( Sun,Mon,Tue,Wed,Thu,
    Fri,Sat,Nul) ;
```

```
type GENDER is
  ( Female, Male, Nul ) ;
```

```
type GENDER_MIX is record
  This_Date
    : Date.DATE ;
  Percent_Women
    : NATURAL ;
end record ;
```

```
type LOCATION is new
   STRING(1..10) ;
function Close
--   Returns the location
-- that is nth closest
-- (i.e. least expensive
-- travel cost) to the
-- entered location.
  ( Origin
      : LOCATION ;
    Ordinal
      : NATURAL )
 return LOCATION ;
```

```
type PROXIMITY is record
  Origin
    : Location.LOCATION ;
  Destination
    : Location.LOCATION ;
  Order
    : NATURAL ;
end record ;
```

```
type SERIES is record
  BT : BT_Class.BT_CLASS ;
  ST : ST_Class.ST_CLASS ;
end record ;
```

```
type ST_CLASS is record
  Start
    : Date.DATE ;
  Place
    : ST_Facility.ST_FACILITY ;
  Course
    : ST_Course.ST_COURSE ;
  Seats_Taken
    : NATURAL ;
end record ;
```

```
type ST_COURSE is
 (ST1,ST2,ST3,ST4,ST5,ST6,Nul);
```

```
type ST_FACILITY is record
  Classroom
    : NATURAL ;
  Place
    : Location.LOCATION ;
  Capacity
    : NATURAL ;
end record ;
```

```
type STUDENT is record
  The_SSN
    : SSN.SSN ;
  BT_Scheduled
    : BT_Class.BT_Class ;
  ST_Scheduled
    : ST_Class.ST_Class ;
end record ;
```

```
type WEIGHT is
  ( Light, Heavy, Nul ) ;
```

```
type WEIGHT_MIX is record
  Date          : Date.DATE ;
  Percent_Heavy : NATURAL ;
end record ;
```

The definition of BT_Class and ST_Class are very similar. Components for date, location, and seats taken are identical. These could be set up as a package called Class in which the exported type is a discriminant type based on case BT or ST. However, the executable code for the operations that will be needed for BT classes are very different from those of ST classes. Therefore, keeping them distinct types and packages further organizes the eventual implementation.

Director Object

The governing director object can be written in complete Ada syntax based on the requirements statement. In order to bridge the many functional gaps that exist before the design is performed, classes and operations that do not exist yet are referenced in the director object. They serve as a starting point for further design class identification.

```
with Date, Dates, Gender;
with Series, Series_List ;
with SSN ;
with ST_Courses ;
with Student ;
with Text_IO ;
with Weight ;

procedure Reserve is

Bad_Dates  : Dates.DATES ;
Good_Dates : Dates.DATES ;
Selection  : Series.SERIES ;
Start      : Date.DATE ;
The_Series
  : Series_List.SERIES_LIST ;
These_ST
  : ST_Courses.ST_COURSES ;
This_BT     : Weight.WEIGHT ;
This_Gender : Gender.GENDER ;
This_Weight : Weight.WEIGHT ;
This_SSN    : SSN.SSN ;

begin

Student.Prompt_For_Requirements
  ( Student_SSN
      => This_SSN,
    Student_Gender
      => This_Gender,
    Desired_BT
      => This_BT,
    Weight_Capability
      => This_Weight,
    Desired_ST_Courses
      => These_ST,
    Date_First_Available
      => Start,
    Dates_Unavailable
      => Bad_Dates ) ;
```

```
-- Build the list of dates on
-- which the student is
-- available for training.
Good_Dates
 :=
Dates.Build_Class_Start_Dates
  ( First
      => Start,
    Unavailable
      => Bad_Dates ) ;

-- Build the list of
-- training offered.
The_Series
 := Series_List.Build
     ( BT
         => This_BT,
         -- ... Whether
         -- light or
         -- heavy.
       This_Gender
         => This_Gender,
       Prioritized_ST
         => These_ST,
       Available_Dates
         => Good_Dates ) ;

if Series_List.Length
    ( Of_This
        => The_Series ) > 0 then
-- if training choices were
-- found
  Series_List.Display
   ( This => The_Series ) ;

  Series_List.Select
   ( This => Selection ) ;

  Series.Reserve
   ( This => Selection ) ;

  Student.Record_Reservation
   ( Student_SSN
       => This_SSN,
     Training
       => Selection ) ;

else
  Text_IO.Put_Line
   ("No choices available.") ;
end if ;
end Reserve ;
```

Functional List Class Design

A training series is made up of BT and/or ST. The software builds a list of training series. This list is an object and a functional list class will be defined for it. In order to build this list, a list of BT classes and/or a list of ST classes are built and evaluated. These are functional list objects also, with their own classes. A number of lists are needed to build a list of BT classes, including: list of dates on which the student is not available (input by the user); list of valid locations for the weight category desired; and list of dates for which the student meets the gender mix requirements. These are functional list classes as well. The ST class list is constrained by the fact that it must come after BT if BT is being taken and is limited to the list of courses being considered. This is an operation exported by the ST date list class. The ST date list that is built is further constrained by the list of unavailable dates, list of dates for which the student meets the gender-mix requirement and, if the ST does not require heavy lifting, by the list of dates for which the student meets the weight-mix requirement.

Many of the important functional list classes are specified below.

The Series_List functional list package provides the ability to build and manipulate training series lists. The body of Series_List.Build will consist of a call to BT_Classes.Build and ST_Classes.Build and will contain the logic to pair up BT and ST and sort the resulting list based on lowest travel cost criteria.

```
with ... ;
package Series_List is

--  This functional list
-- package
-- provides the ability to
-- build and manipulate
-- lists of training series.

package Series_List_Package
  is new Linked_List
         ( Element
            => Series);

type SERIES_LIST is new
   Series_List_Package.LIST ;

function Build
  ( Gender
     : in Gender.GENDER ;
    Weight_Category
     : in Weight.WEIGHT ;
    BT
     : in Weight.WEIGHT ;
    ST
    : in ST_Courses.ST_COURSES;
    Available_Dates
     : in Dates.DATES )
return SERIES_LIST ;

function Combine
  ( BT_Need
     : in Weight.WEIGHT ;
    BT_List
    : in BT_Classes.BT_CLASSES;
    ST_Need
     :in ST_Courses.ST_COURSES;
    ST_List
    : in ST_Classes.ST_CLASSES)
 return SERIES_LIST ;

procedure Sort_By_BT_Start_Date
  ( This : in out SERIES_LIST);
procedure Sort_By_Proximity
  ( This : in out SERIES_LIST);

procedure Find
  ( BT_Start
     : in     Date.DATE ;
    In_List
     : in     SERIES_LIST ;
    Position
     :    out INTEGER;
    Count
     : out NATURAL) ;
end Series_List ;
```

```
with ... ;

package BT_Classes is

package BT_Class_List is new
  Linked_List
    (Element
       => BT_Class.BT_CLASS ) ;

type BT_CLASSES
  is new BT_Class_List.LIST ;
```

```
function Build
  ( Gender
      : in Gender.GENDER ;
    Weight
      : in Weight.WEIGHT ;
    These_Dates
      : in BT_Dates.BT_DATES ;
return BT_CLASSES ;

function Dates
  --   For a list of BT
  -- classes,
  -- return a merge sorted
  -- list of start dates.
  ( Classes
      : BT_CLASSES )
 return Dates.DATES ;

procedure Sort_By_Date
  ( This : in out BT_CLASSES );

procedure Sort_By_Location
  ( This : in out BT_CLASSES );

procedure Find
  ( Start
      : in Date.DATE ;
    In_List
      : in    BT_CLASSES;
    Position
      :    out INTEGER ;
    Count
      :    out NATURAL);

procedure Find
  ( Location
      : in  Location.LOCATION ;
    In_List
      : in     BT_CLASSES ;
    Position
      :    out INTEGER ;
    Count
      :    out NATURAL ) ;
end BT_Classes ;
```

```
with ... ;

package ST_Classes is

package ST_Class_List is new
  Linked_List
    ( Element
        = > ST_Class.ST_CLASS );

type ST_CLASSES
  is new ST_Class_List.LIST ;

function Build
  ( Courses
   : in ST_Courses.ST_COURSES;
   These_Dates
     : in ST_Dates.ST_DATES )
return ST_CLASSES ;

function Dates
  --   For a list of ST
  -- classes,
  -- return a merge sorted
  -- list of start dates.
  ( Classes
      : ST_CLASSES )
 return Dates.DATES ;

procedure Sort_By_Date ...;
procedure Sort_By_Preference
    ... ;
procedure Find_Date ...;
procedure Find_Preference ...;
end ST_Classes ;
```

```
package Dates is
type DATES is new
  List_Structure
    ( ELEMENT
        = > Date.DATE ) ;
end Dates ;
```

```
package BT_Dates is

type BT_DATES is new
  Dates.LIST ;

function Build
 ( ... )
 return BT_DATES ;

function Dates_To_BT
 ( These_Dates
    : in Dates.DATES )
 return BT_DATES ;

function BT_To_Dates
 ( These_BT_Dates
    : in BT_DATES )
 return Dates.DATES ;

function Next_Week
 ( BT_Dates
    : BT_DATES )
 return Dates.DATES ;
end BT_Dates ;
```

```
package ST_Dates is

type ST_Dates is new
 Dates.LIST ;

function Build
 ( ... )
 return ST_DATES ;

function Dates_To_ST
 ( These_Dates
    : in Dates.DATES )
 return ST_DATES ;
function ST_To_Dates
 ( These_ST_Dates
    : in ST_DATES )
 return Dates.DATES ;
end ST_Dates ;
```

Module Executable Code

The executable code of the bodies of the operations are largely composed of calls to operations. The complex processing logic is spread out. There will be some operations that bear the responsibility of major logic, but they will be focused on a single objective. These modules typically have a cyclomatic complexity of 8-10.

During the course of writing executable code, the need for various utility functions in fundamental and design classes arises. The appropriate classes are then expanded with the necessary capabilities. There is a necessary overhead in data processing when distinct functional list classes are used. Types and lists must be converted before use in foreign operations. This string typing overhead is justified when it results in code that can be readily verified for functional correctness.

The following example is presented to illustrate functional list code characteristics.

```
separate (Series_List)

function Build
 ( Gender
    : in Gender.GENDER ;
   Weight_Category
    : in Weight.WEIGHT ;
   BT
    : in Weight.WEIGHT ;
   ST
   : in ST_Courses.ST_COURSES;
   Available_Dates
    : in Dates.DATES )
return SERIES_LIST is
```

```
-- local variables

begin

if BT /= Weight.Nul then
  These_BT_Dates
    := BT_Dates.Dates_To_BT
        ( These_Dates
            =>Available_Dates);
  The_BT_Classes
    :=  BT_Classes.Build
        ( Gender
            => Gender,
          Weight
            => BT,
          These_Dates
          => These_BT_Dates);
end if ;

if ST_Courses.Length
    ( Of
        => ST ) > 0 then
-- Set up ST start dates
-- based on whether BT is
-- taken.
  if BT /= Weight.Nul then
    Start_Dates
      := BT_Dates.Next_Week
          ( BT_Dates
          => These_BT_Dates );
    ST_Start_Dates
      := ( ST_Dates.Dayes_To_ST
            ( These_Dates
                => Start_Dates);

  else -- BT was not taken
    ST_Start_Dates
      := ST_Dates.Dates_To_ST
          ( These_Dates
            =>Available_Dates);
  end if ;

  The_ST_Classes
    := ST_Classes.Build
        ( Courses
            => These_ST,
          These_Dates
            =>ST_Start_Dates);

end if ;

The_Series
  := Series_List.Combine
      ( BT_Requested
          => BT,
        BT_List
          => The_BT_Classes,
        ST_Requested
          => ST
        ST_List
          => The_ST_Classes);

return The_Series ;
end Build ;
```

DISCUSSION

A hallmark trait of object-oriented development is deferred gratification. This refers to the fact that identified requirements are rarely implemented directly. The first task is to define the classes needed. Designers then imbue each class with all needed properties and capabilities. The glue of an application program will then utilize those operations it needs to accomplish its goals. Different applications will use different operations of the same class. Thus, the value of some of the immediate work may not be realized until later projects. The effects of deferred gratification partially accounts for the observation that first projects take longer to develop in Ada. Most development time is spent defining all the

many varied aspects of classes and some of the operations defined will not be used in the current project.

This property of object-oriented methods makes it extremely important to speed-up development in any way possible. In order to make the process of class identification and definition effective and efficient, software engineers need a library of techniques which they can consult when searching for an appropriate design approach. The functional list is just such a technique. It should be used when deemed pertinent to the problem domain.

CONCLUSION

Functional list classes divide complex processes into fundamental operations that are easy to code and test. One reason why functional lists are easy to code is that they are directly based on generic list structures, generic sorts and generic searches. Much work has been done on lists and related utilities, and, consequently, tested code is readily available. The functional list concept builds on this work and takes it a step further into the realm of solving domain-specific problems.

REFERENCES

1. Anderson, K. J., Beck R. P., & Buonanno T. E. Reuse of software modules. AT&T Technical Journal, July/August 1988, 67, 71-76.

2. Basili, V. R., & Perricone, B. T. Software errors and complexity: An empirical investigation. Communications of the ACM, 1984, 27, 42-52.

3. Booch, G. Software Engineering with Ada. Menlo Park, CA, Benjamin/Cummings, 1986.

4. Colbert, E. The object-oriented software development method: A practical approach to object-oriented development. Proceedings of TRI-Ada'89, 1989, 400-415.

5. Danforth, S., & Tomlinson, C. Type theories and object-oriented programming. ACM Computing Surveys, March 1988, 20, 29-72.

6. Genillard, C., Ebel, N., & Strohmeier, A. Rationale for the design of reusable abstract data types implemented in Ada. Ada Letters, March/April 1989, 9, 62-71.

7. Lew, K. S., Dillon, T. S., & Forward, K. E. Software complexity and its impact on software reliability. IEEE Transaction on Software Engineering, 1988, SE-14, 1645-1655.

8. Lind, R. A., & Vairavan, K. An experimental investigation of software metrics and their relationship to software development effort. IEEE Transactions on Software Engineering, 1989, 15, 649-653.

9. Martin, J., & McClure, C. Structured Techniques: The basis for CASE, Englewood Cliffs, NJ, Prentice Hall, 1988.

10. McCabe, T. J., & Butler, C. W. Design complexity measurement and testing. Communications of the ACM, 1989, 32, 1415-1425.

11. Meyer, B. Reusability: The case for object-oriented design. IEEE Software, March 1987, 4, 50-64.

12. Meyer, B. Object-oriented software construction. Great Britain: Prentice Hall International, 1988.

13. Perez, E. P. Simulating inheritance with Ada. Ada Letters, September/October 1988, 8, 37-46.

14. Shen, V., Yu, T., Thebaut, S., & Paulsen, L. Identifying error prone software - An empirical study. IEEE Transactions on Software Engineering, 1985, SE-11, 317-324.

15. Whitcomb, M. J., & Clark, B. N. Pragmatic definition of an object-oriented development process for Ada. Proceeding of TRI-Ada'89, 1989, 380-399.