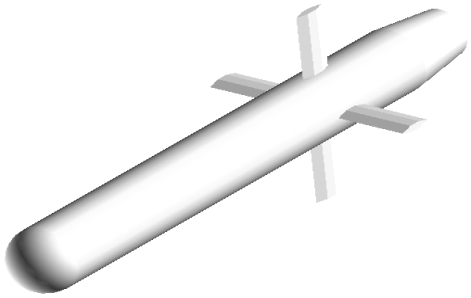
AML
Basic Training
Manual

V3.06



TECHNOSOFT INC.

TechnoSoft AML Basic Training

Manual: **Version 3.06**

Techno**S**oft **I**nc.

11180 Reed Hartman Highway

Cincinnati, OH 45242

1. Introduction

1.1 Training Manual Organization, Syntax and Style

This Manual is an introduction and overview to TechnoSoft's Adaptive Modeling Language (AML) for designers and engineers. TechnoSoft planned it for AML student/developer use in conjunction with instructor lecture and hands-on exercises covering AML applications. Like so many software tools, hands-on experience makes textual descriptions more meaningful. With an instructor, students can ask questions and get help as needed.

AML has many more facilities than those described in this Manual. Once students master and understand exercises included in this Manual, they will be better able to understand and use other AML extensions given in the AML Reference Manual to develop their applications.

After covering an introduction to modeling, object architecture, and AML, the instructor will introduce each exercise. After the instructor's explanation, students should attempt to work the exercise on their own. Anyone needing assistance can receive it from the instructor on an individual basis.

1.2 AML Introduction

1.2.1 AML Modeling

Adaptive Modeling Language (AML) is a modeling language for concurrent engineering. AML provides a paradigm for modeling and organizing vital engineering knowledge required for integrating and automating entire engineering cycles from design to production.

AML is based on the concept of object-oriented programming. In object-oriented programming, the building blocks of applications are objects; they are not procedures or functions. In this introduction, the principles of object-oriented programming and the advantages of the methodology over traditional programming techniques (procedural, modular) are introduced.

1.2.2 Why object-oriented Methodology?

Most software today is designed and implemented to solve a certain problem. The programmer starts by examining the task to be performed and develops a strategy to deal with the task using subroutines and procedures that don't reflect the physical world. This makes code maintenance quite difficult and inefficient. With object-oriented programming, the programmer starts by examining the aspects of the real world that need to be modeled in order to perform the task. The models developed use objects that reflect the physical world. Because the structure of the object-oriented software reflects the real world, conceptualization, maintenance, and modification of the software can be easily performed.

Using object-oriented technology, one can construct a model of some aspect of a company's operation. The model reflects real world entities and operations and can be used to solve a number of related problems (reuse).

Object-oriented Methodology relies on the following mechanisms or principles

- Objects
- Classes
- Inheritance
- Methods

Understanding these mechanisms will lead to an understanding of the object-oriented paradigm.

1.2.2.1 Objects/Subobjects

The basic building units of the object-oriented approach are objects. An object, in this paradigm, is a uniform representation of a real world entity. In the object-oriented approach, the programmer thinks in terms of physical world objects and the new data structures (objects) are defined in terms of real world objects. Objects can be thought of as data abstractions that contain a collection of related data elements (properties and sub-objects) and a set of procedures (methods) that operate on the object elements. Objects have the following essential features:

- Objects interact with one another; this is achieved by message passing. Message passing in AML is basically a call to a method that is associated with the object to be communicated with. In AML an object can communicate with another object by changing the value of an attribute that is associated with the object to be communicated with.
- Every object in the system has a unique identity. Object identity is the property in the object that distinguishes the object from all other objects in the system. In an object-oriented system the object identity is unique and independent of the value of the object attributes.
- In an object-oriented approach, the units of encapsulation are objects. An object encapsulates state information (data) and behavior (operations). Operations are just a way of changing the state of an object.
- If an object is logically related to one or more objects, then there is an association between the objects. Associations can be implemented using attributes or by using an object to represent the association.
- It is possible to build composite objects. A composite object is one which consist of parts which themselves are objects (object-subobject relation). For example a car is made of doors, body panels, frame, windshields, etc., where each part is an object.

The following is an example of problem decomposition where the user is asked to build a model of a desk using an object-oriented programming language. The user can break the problem up according to Figure 1 where the desk has a top panel, left panel, right panel, and a modesty panel. After the objects that need to be modeled are identified the user can start working on the model.

Simple part hierarchy (object-subobject)

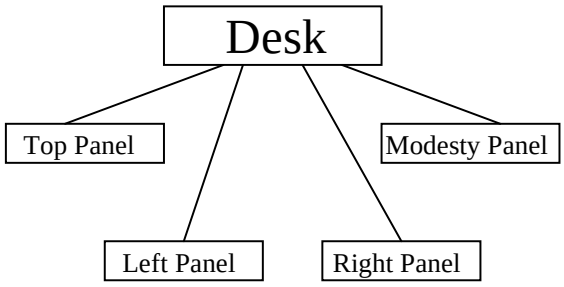


Figure 1

In the desk example, the panel objects are all basically the same. It would be a waste of time and effort to define each panel separately. It is more efficient to define a generic panel and instantiate the generic panel to reflect the state of a specific panel.

1.2.2.2 Classes

In the object-oriented paradigm the tool for creating new data types is the class. A class can be thought of as a template that is used to create objects (e.g., class is a recipe for making a cake, while an object is a cake that was made using the recipe). The objects belonging to a particular class are said to be instances of the class.

Classes allow objects to be defined in a very efficient manner. The methods and variables for a class are defined once, in the class definition. Each instance of the class contains the actual value of the variables. The following concepts are essential to understanding classes:

A class defines the structural definition of instances of the class. The class defines the names of attributes (state) and methods (behavior) of an object belonging to this class.

Classes are used to create objects (instances of a class). An object belongs to exactly one class, while a class can have a number of instances.

Inheritance is a very important mechanism for class definition. A class can be defined in terms of existing classes establishing a superclass - subclass relationship. A subclass inherits the attributes and operations of the superclass(es) and can add attributes and operations. A subclass can be thought of as a specialization of the superclass. For example, assume that a user is to model a number of cars (sedan, coupe, hatchback, ...). These cars have the same basic features (4 wheels, front windshield, ...). To model the system the user can define a superclass called AUTOMOBILE that the rest of the classes inherit from. The simple class hierarchy is illustrated in Figure 2.

Simple class hierarchy (superclass-subclass)

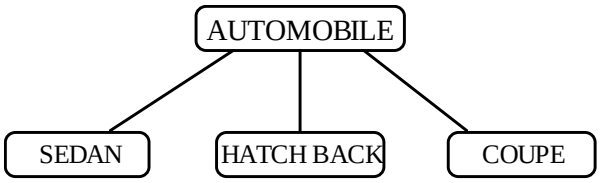


Figure 2

A class can inherit from more than one class. This is referred to as multiple inheritance. This is the case when a class of objects has to play multiple roles. For example, a dolphin class can inherit from fish class and mammal class. A foreman of a site may play a dual role, functioning both as a supervisor and as a builder. When multiple inheritance occurs a tree-like structure can be developed to describe the class hierarchy (see Figure 3). This class hierarchy is important in resolving conflicts; conflicts occur because superclasses can have the same attribute and/or operation names. The conflicts have to be resolved before generating the final class definition.

Multiple inheritance, Dolphin has properties of Fish and Mammal

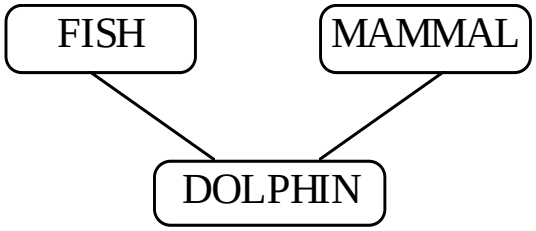


Figure 3

In the Desk example, the programmer is likely to create a panel class that inherits from a BOX class (assume that there is a predefined class called BOX). All the panels will be object instances of the panel class (all panels have the same basic shape).

class PANEL
inherit-from BOX
properties

width 10.0
height 8.0
depth 1.0

The panel class can then be used to define the panels in the desk.

class DESK
inherit-from .....
properties
.
.
subobjects
{left-panel **class** panel
width ..
height ..
depth ...
orient appropriately
}
{right-panel **class** panel
width ..
height ..
depth ...
orient appropriately
}

1.2.2.3 Methods

In the object-oriented approach, the basic building blocks are objects. The objects that make a model need to communicate with one another by calling methods (operation) that are associated with the object to be modified. In object-oriented terminology objects communicate with one another through messages. A message is simply the receiving object combined with the name of one of its methods. One of the advantages of using methods is that they allow for name reuse (overloading).

In AML a different class represents each kind of geometric shape. Through overloading, the user can use the same name for the drawing method in every class. In AML, to draw a graphic object all the user has to do is invoke draw on the object to be drawn. So to draw line-1 (line-1 is an object of type line) the user invokes (draw line-1), to draw an arc the user invokes (draw arc-1) etc. So if the user defines a new class LENS, the user can define a draw method on the class. This allows user defined classes to behave in the same manor as system defined classes. If overloading was not available, then the user would need to give a different operation name for drawing objects of different classes. Thus, to draw a line, the user may have a draw-line operation, to draw an arc a draw-arc operation maybe required and so on. Depending on the object class, the user will then have to call the appropriate operation. Name reuse (overloading) allows for the design of simple and elegant code.

1.2.3 How Can AML Enable a Developer?

TechnoSoft designed AML to support several specific functions. AML provides an extended modeling foundation that allows designers to model a range of physical and non-physical factors within a single AML model. For example, an AML model can carry object and component costs as object properties so designers can directly determine how design decisions affect costs. Also, manufacturing processes can be included as objects within AML models. Objects can have properties linked together in complex dependencies. When a property of an object is changed, AML automatically notifies all other properties dependent on that property so they are consistent with the new value when re-demanded. AML can interface model design parameters to software packages outside AML. For example, a designer can interface a life cycle estimation software package with his AML design to compute the part's anticipated life as a property of the part. The designer can interface the expected life property to a cost model.

AML includes the following functions to support designers.

- An object architecture-modeling paradigm allows designers to model in familiar terms.
- Full supported and portable between UNIX and Windows platforms (running natively)
- Single underlying object oriented architecture
- Open architecture for foreign applications' seamless integration
- Innate UI builder
- Common syntax throughout the different modules
- Real time dependency tracking
- Demand driven computation
- Full support of IGES/STEP/DXF
- Support of various geometric Modelers with full model compatibility
- Dynamic objects and model builder
- Mixed wireframe, surface, and solid modeling
- Automatic dimensioning and detailing

- Analysis modeling and meshing
- Activity based cost and operational modeling
- Model configuration management and visualization
- Distributed and collaborative modeling over a network of heterogeneous machines

In summary, AML is not limited to creating geometrical designs. AML is a highly usable design tool that designers can address physical aspects of part design as well as broader problems related to physical design.

1.3 Training Manual Syntax and Style

The format and style of the text in this manual will represent different things. The *italicized* text is used to differentiate AML code from English words within normal text. Example code is represented with the following fonts:

```
Example code in courier font.
Return values from AML given in bold courier font.
```

This manual presents AML programming through a case study. Each section of the case studies has the following format:

- Title of the current step in the model creation,
- Brief explanation of the part model and its purpose in the model,
- New AML constructs used in creating this model part,
- Exercises and,
- Proposed solutions to the exercises with
    - example code and
    - code explanation.

✓ Special topics or statements are highlighted with the checkmark symbol.

Each new object or AML construct is explained, with examples, before the code is given. The code is developed step by step, making small advances to finally complete a certain objective. (Please note that the new AML constructs may have more properties than are given in this manual.) The files build upon each other, changing slightly, while introducing new topics and more efficient methodologies of programming in AML. It is very important to create these files in AML, and create the model while following along with the manual. Each file builds upon the previous file, therefore minimal typing is involved. The best way to learn AML is to practice writing and using AML. Follow along in the examples and focus on understanding each step before moving on to the next exercise.

2. Missile Case Study – AML Modeling Practices and Geometry Creation

2.1 Explanation

The case study used in this manual shows the functionality of AML through examples. The study demonstrates the basic process by which a developer/engineer solves a problem using geometric reasoning and software integration with AML.

The missile case study involves modeling a simple missile-like air vehicle. The training manual shows *class*, *function* and *method* syntax and usage with application to the missile design development. The goal is to produce a model that could be used for simple design simulation, analysis, and "producability" assessment. The developer models the geometry of the missile and learns how to integrate the geometry with other elements of the design process. The training manual and instructor give simple examples to augment these topics.

2.2 The Final Geometry

Figure 4 shows the final geometry of the missile geometry model.

Final geometry

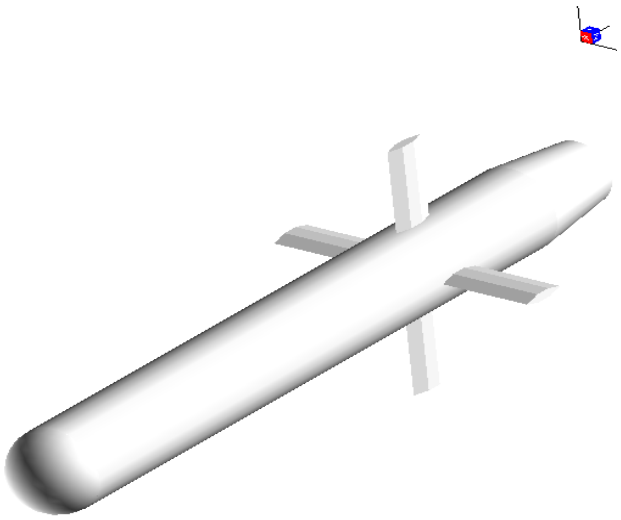


Figure 4

By the end of this case study, you will be able to create the geometry in Figure 4, set up a parametric Finite Element Mesh and Analysis, publish the model in a standard format for light-weight distributed modeling and visualization, manage source code, all enabling the automatic design and analysis of this model in a matter of seconds!

The following sections develop each component in the missile model (the coordinate systems, nose, body section, aft section, fins), tag, mesh, analyze and publish the model for off-line collaboration and visualization.

2.3 The Missile Geometry Class with Coordinate Systems

Each missile component is positioned on the overall missile geometry using a coordinate system. Figure 5 shows these coordinate systems in their final positions with respect to an overall missile coordinate system. This overall missile coordinate system serves as a base reference for the entire missile geometry. For example, a user may take this missile geometry and position it on an airplane model using this coordinate system for a reference point.

Missile Coordinate Systems

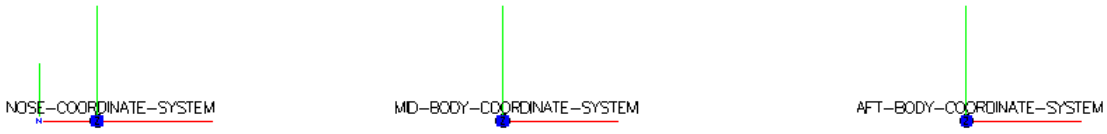


Figure 5

Note that the *missile-coordinate-system* is not labeled in Figure 5.

New AML Constructs

This section covers the AML construction syntax for defining a class and/or object. In addition, the following AML concepts and constructs are used:

- evaluating expressions
- *list, quote*, quoted lists
- *object*
- *define-class*
  - specifying inheritance
  - specifying properties
  - specifying subobjects
- *coordinate-system-class*
- *create-model*
- starting and exiting AML
- overall model development procedures

Opening Comments

- AML is not case-sensitive.
- Data typing is not verified upon compilation in AML, but some data types are expected such as a string, symbol, list, number, or an object.
- AML is both an interpreted and compiled language.

EXPRESSIONS

AML uses prefix notation to evaluate expressions. That means that the operator appears first followed by the arguments. Evaluation of the arguments occurs before the arithmetic operation. This allows embedding of calculations within other calculations, also known as nesting. AML uses typical coding notations for multiplication (*), division (/), addition (+), and subtraction (-). Many other arithmetic, trigonometric, and vector and matrix functions are available in AML. Please see the AML Reference Manual for further documentation of these functions.

Examples:

```
AML> (- 1 2)
-1
AML> (* 3 4)
12
AML> (/ 4 -2)
-2
AML> (+ (- 1 2) (* 3 4))
11
```

The entire syntax for the last example above is an expression that returns the value of 11, and the arguments *(- 1 2)* and *(* 3 4)* are expressions that return -1 and 12, respectively. The arguments -1 and 12 are also considered expressions which return themselves.

LISTS

A list is a collection of elements. Examples of lists are: (1 2 3), (a b c), ("Bob" "Jim" "Steve"), (1 a "Bob"). A list may be specified in any of the following three formats.

Format:

(list arg1 arg2 ... argn)
or
(quote arg1 arg2 … argn)
or
‘(arg1 arg2 … argn)

Creating a list:
'(1 2 3) and (list 1 2 3) both create a list containing integer elements of 1, 2, and 3. However, the two means of specifying the list are not identical. Using the function *(list ...)* causes each element in the list to be evaluated. When using *‘(…)* , AML will not evalutate the elements inside of the parenthesis. The example *(list 1 2 3)* returns the same result as *'(1 2 3)* because each of the elements evaluates to themselves. The example *(list (* 1 2) (- 2 3) (+ 3 4))* returns (2 -1 7) because AML evaluated each element inside the list.

Examples:

With p1 defined as 1,
p2 defined as 2,
p3 defined as 3,

```
AML> '(p1 p2 p3)
(P1 P2 P3)
AML> (list p1 p2 p3)
(1 2 3)
AML> (list
        (+ p1 p2 p3)
        (- p1 p2 p3)
        (* p1 p2 p3)
        (/ p1 p2 p3)
        )
(6 -4 6 1/6)
```

Some list extraction and query functions are: **first, rest, last, nth, and length**.

(first list)
(rest list)
(last list)
(nth index list)
(length list)

Note: *nth* returns the element in the list in the position specified by the index. The index of the first element is 0 (zero). If the index goes beyond the length of the list, the result nil is returned. The function *first* returns the first element in the list. It is equivalent to *nth* with an index of 0 (zero).

Note: The functions *rest* and *last* both return lists. *Rest* returns everything except the first element in the list. *Last* returns a list containing only the last element in the list.

Examples:

```
AML> (first  '(a b c))
A
AML> (rest   '(a b c))
(B C)                                   ;;; Note:  This is a list.
AML> (last   '(a b c))
(C)                                     ;;; Note:  This is a list.
AML> (first (last '(a b c))
C
AML>  (nth 0 '(a b c))
A
AML>  (nth 1 '(a b c))
B
AML>  (nth 1 (nth 0 (list '(1 2 3) "hello")))
2
AML>  (nth 3 '(a b c))
NIL
AML>  (length '(a b c))
3
```

OBJECT [Class]

Object is the highest level class from which user-defined classes should inherit. While developing AML source code, classes should not inherit from any class that is a super-class of *object*.

Object is primarily used to define classes which will not have any associated geometry or graphic representation. Often, this class is used to define classes which will be mixed with other classes when defining objects. Most of the predefined classes in AML have *object* as a super-class.

DEFINE-CLASS [Construct]

Define-class, a fundamental AML construct, is used to describe the structure of new classes. All predefined and user-defined AML classes are defined using this construct, including those in the user interface, meshing, and analysis modules of the AML syntax. In any *define-class*, the following may be specified:

1) The class or classes from which the new class should inherit (superclasses). This is required for all new class definitions.

2) The properties of the new class and their formulas (attributes).

3) The subobjects of the new class (children).

The properties and subobjects given in the class definition add to those that exist in the definitions of the classes super-classes. Properties and subobjects from the super-classes will be replaced (overwritten) if they have the same name as any given in the new class definition.

Once defined using the *define-class* construct, instances of new classes can be created. Creating an instance is done using the *create-model* and *add-object* commands. *Create-model* will create a new instance of the class and make that instance the top of a new model (root object of the model). *Add-object* is used to create an instance of a class as a subobject of an object that already exists.

Format:

```
(DEFINE-CLASS class-name
:inherit-from ()
[:properties (property-specifications)]
[:subobjects (subobject-specification-lists)]
)
```

Arguments:

class-name	Any symbol giving the name of the class being defined.
:inherit-from	A list of predefined classes this class inherits from.
:properties	A list of property specifications. Each specification may be a simple property/formula pair or an object specification list containing the name, class, and property/formula pairs for the property object.
:subobjects	A list of subobject specification lists containing the name, class, and property/formula pairs for each subobject.

Notes:

- The class used in the specification for a subobject or a property object must be the name of a pre-defined class. **The code will not compile if the class has not been previously defined**!
- Using a class name as a subobject name or a property name can be confusing and should be avoided.
- All properties and subobjects must be **uniquely named** within the scope of a class definition.

Example:

```
(define-class EXAMPLE-SUBOBJECT-CLASS
    :inherit-from (object)
    :properties (
        id              "OU812"
        part-number     2.0
    )
)

(define-class EXAMPLE-MODEL-CLASS
    :inherit-from (object)
    :properties (
        model-name      "Model For AML Basic Training"
    )
    :subobjects (
        (sub-1 :class 'example-subobject-class
            part-number     3.0
        )
        (sub-2 :class 'example-subobject-class
            id              "ZZ184"
        )
    )
)
```

The subobjects refer to their "container object" as their **parent** or **superior**. The *example-model-class* is the superior of *sub-1* and *sub-2* and *sub-1* and *sub-2* are condidered **children** of *example-model-class*.

Specifying Inheritance

Inheritance is a mechanism for class reuse. Through inheritance, a class will have all of the same properties, subobjects, and methods as the classes that it inherits from (its superclasses). The *:inherit-from* section of the *define-class* construct accepts a list of classes to be used as superclasses. If a property, subobject, or method is present in more than one of the superclasses, the order of precedence is from left to right.

Example:

```
(define-class MATERIAL-CLASS
    :inherit-from (object)
    :properties (
        material        'wood
        density         0.0
        )
    )

(define-class EXAMPLE-SUBOBJECT-CLASS
    :inherit-from (material-class)
    :properties (
        id              "OU812"
        part-number     2.0
        material        'steel
        )
    )

(define-class EXAMPLE-MODEL-CLASS
    :inherit-from (object)
    :properties (
        model-name      ”Model For AML Basic Training”
        )
    :subobjects (
        (sub-1 :class 'example-subobject-class
            part-number     3.0
            )
        (sub-2 :class 'example-subobject-class
            id              "ZZ184"
            material        'aluminum
            )
        )
    )
```

Specifying Properties

The optional *:properties* section of the *define-class* construct is used to assign properties and their associated formulas to classes. Properties may be specified as a simple property/formula as shown in the *define-class* example or as a property object specification (explained later in the manual). The properties may be the names of new properties to be added to the class or names of properties that are in the superclasses. If the names are the same as those in superclasses they are considered overriding properties. The property specifications are used as the defaults during the creation of instances.

✓ The formula of a property can be a function call, method call, any calculation, or simple value that gets evaluated. These include functions to run programs, or even making calls to Fortran or “C”.

Specifying Subobjects

The optional *:subobjects* section of the *define-class* construct must be a list of subobject specifications. The subobjects are created as instances that are children of the class being defined. The form of the subobjects requires the following format:

(subobject-instance-name :class class-to-be-instantiated property-specifications)

The subobject-instance-name is any symbol that will become the name of the instance that is created from the specification. The class-to-be-instantiated may be any expression that evaluates to a class name in symbol form.

The property specifications at the subobject level are exactly the same as the *:properties* level for *define-class*.

COORDINATE-SYSTEM-CLASS [Class]

The *coordinate-system-class* provides a cartesian orthogonal reference frame that can be used to position other objects in a model. It adds a visible indicator of the coordinate frame position and orientation and is used as a reference frame for other objects (including other *coordinate-system-class* objects). A *coordinate-system-class* object is drawn as a set of axes in the local x, y, and z directions; a box displaying x, y, and z; and the name of the coordinate-system-class object. These components can be individually turned off. The z-axis (*vector-k*) is assumed to be the cross product of *vector-i* and *vector-j*.

Properties:

origin	The origin property specifies the position of the origin of the *coordinate-system-class* object. Defaults to '(0 0 0).
vector-i	The direction of the x-axis of the *coordinate-system-class*. Defaults to '(1 0 0).
vector-j	The direction of the y-axis of the *coordinate-system-class*. Defaults to '(0 1 0).

Examples:

```
(define-class ANGLED-COORDINATE-SYSTEM-CLASS
    :inherit-from (coordinate-system-class)
    :properties (
        origin   '(3 1.03 6)
        vector-i '(1 1 0)
        vector-j '(-1 1 0)
        )
    )

(define-class SET-OF-COORDINATE-SYSTEMS-CLASS
  :inherit-from (object)
  :subobjects (
              (coord-sys-1 :class 'coordinate-system-class
                           )
              (coord-sys-2 :class 'angled-coordinate-system-class
                           )
              (coord-sys-3 :class 'coordinate-system-class
                           origin   '(1 2 0)
                           vector-i '(0 1 0)
                           vector-j '(-1 0 0)
                           )
              ))
```

CREATE-MODEL **[Function]**

Create-model allows the user to create a new model (instantiate a class) based on the class given in the *name* argument. This creates a model (an instance, also known as an "object") of a particular class. This instance becomes the current model, also known as the "root" of the tree. In AML, all models are placed in a part hierarchy such that all user defined models are children of the *model-manager*. The *model-manager* has one predefined subobject called *interface* which is the object where all of the AML user interface objects are stored. The *model-manager* is the absolute root of all AML objects. To select the various user defined models, use the function *select-model*.

Format:

(CREATE-MODEL class-name)

Arguments:

class-name	The name for the model being created in symbol form; the name must be a valid pre-defined class name.

Examples:

```
(define-class MATERIAL-CLASS
    :inherit-from (object)
    :properties (
        material        'wood
        density         0.0
        )
    )

AML> (create-model 'material-class)
#<MATERIAL-CLASS @ #x2224280a>
```

See Also:

```
select-model
delete-model
```

Starting AML, Text Editing, and Exiting AML

AML runs natively on both Unix and Windows platforms. TechnoSoft supports AML for Intel based PC machines as well as Hewlett Packard, Sun, Silicon Graphics, and IBM UNIX machines. AML applications are totally portable between the two platforms.

2.3.1 Starting AML:

For the Windows based machines, double click on the AML icon on the desktop (if available), or find the AML Program Group from the Windows "Start" | "Programs" | "AML" button. Options are provided to start AML with XEmacs (typically for development use) or from a standard

command prompt window (typically for run-time use). If the XEmacs option is selected, the XEmacs application will start for text editing purposes. Clicking on the "Run AML" button will start the AML active command prompt buffer inside the XEmacs environment.

2.3.2 AML Text Editing with XEmacs:

GNU XEmacs is a free, portable, extensible text editor. Free means that everyone may use and redistribute it without a licensing fee. Portable means that it runs on many machines under many different operating systems. Extensible means that you can customize all aspects of its usage (key bindings, fonts, colors, windows and menus). TechnoSoft has customized XEmacs to best suite the AML syntax and has incorporated an active command prompt in the XEmacs environment.

Notation: This section uses standard XEmacs notation to describe keystrokes:

C-x	x represents a key, depress both the control key and x at the same time.
M-x	depress both the meta (also known as "Alt") key and x at the same time.
C-M-x	depress the control key, the meta key and x at the same time.
RET	The return key.
SPC	The space bar.
ESC	The escape key.

Files, Buffers and Windows: XEmacs has three intimately related data structures:

Files: A file is the actual file on disk. You are never editing a file. Rather, you read a copy into XEmacs to initialize a buffer and write a copy of a buffer to a file to save it.

Buffers: The buffer is the basic editing unit. One buffer corresponds to one piece of text being edited. XEmacs can have any number of buffers active at any moment, but only a single buffer selected. This is the buffer that your cursor is in, and where typed commands take effect. A Buffer is deletable and deleting a buffer does not delete the file on disk (though you may lose any editing changes you made if you do not save first).

Windows: A window is a view of a buffer. Due to limited screen space, all buffers may not be viewed at once. You can split the screen, horizontally or vertically, into as many windows as you like and view a different buffer in each window. It is also possible to have several windows viewing different portions of the same buffer. Deleting a window in no way deletes the buffer associated with the window. Each window has its own mode line, but there is still only one minibuffer (the minibuffer is described later).

Mode Line: The last line at the bottom of a buffer is an informational, non removable mode line. It displays important information including:

- The state of the buffer: modified (a pair of asterisks), unmodified (hyphens), or read-only (a pair of % signs).
- The name of the file edited (*scratch* is a buffer available for non file work).
- The major mode (in parentheses).
- The amount of the file seen on the screen:

- All - The entire file.
- Top - The top of the file.
- Bot - The bottom of the file.
- Percentage - NN% indicates the percentage of the file above the top of the window.

The Minibuffer: The blank line below the mode line is the minibuffer. XEmacs uses the minibuffer to display messages. XEmacs also requests input from the user at the minibuffer (it may want you to type yes or no in answer to a question, the name of a file to edit, the long name of a command, etc.).

Help: XEmacs has extensive online help, most of which is available via the help key, **C-h**. C-h is a prefix key. Type C-h twice to see a list of subcommands; type it three times to get a window describing all the subcommands.

Commands to Manipulate Files:

C-x C-f Find-file. Displays a file in a buffer for editing. Execution of this command prompts for the name of the file. If that file is available in another buffer, it switches to that buffer and does not actually read in the file from disk again. If not, find-file creates a new buffer named for the file, and initializes it with a copy of the file. In either case, the current window becomes a buffer containing the contents of the requested file (or current editing of that file). If no file exists, the buffer is named after the file you attempted to find and saving the buffer creates the file.
C-x C-s Save-buffer. Saves a file. More accurately, it writes a copy of the current buffer out to the disk, overwriting the buffer's file and handling backup versions.
C-x w Save-buffer-as. Saves the current buffer to a file. The user is prompted for the file's location.

Commands to Manipulate Buffers:

C-x b Switch-to-buffer. Prompts for a buffer name and switches the buffer of the current window to that buffer. It does not change the window configuration. A new buffer name creates a new empty buffer. The new buffer is empty (even if the new name corresponds to a filename).
C-x C-b List-buffers. Pops up a window that lists all buffers and provides: buffer-name, modification state, size in bytes, major mode and the possible file the buffer is visiting.
C-x k Kill-buffer. Prompts for a buffer name and removes the entire data structure for that buffer from XEmacs. The command provides an opportunity to save a modified buffer. Note that this in no way removes or deletes the associated file, if any.

Commands to Control Display:

C-v Scroll-up. Scrolls forward (towards the end of the file) a windowful or a specified number of lines. By default XEmacs leaves two lines of context from the previous screen.
M-v Scroll-down. Just like C-v, but scrolls backwards.
C-l Recenter. Clears the screen and redisplays, scrolling the location where the

cursor is residing to the vertical center of the screen
C-x 1 Display Single Buffer.
C-x 2 Split screen horizontally.

Undoing Changes:

C-x u Undo. Undo editing, backward in time. XEmacs has infinite undo ability, so that even long chains of commands can be undone. XEmacs has redo capability that allows for reverse direction while undoing, thereby undoing the undo.

Completion, Deleting and Killing:

To save typing, XEmacs offers various forms of completion: this means XEmacs tries to complete partially typed file names, command names, etc. To invoke completion, try typing TAB or SPC.

Emacs provides deletion commands based on the textual objects above. Deletion means to remove text from the buffer without saving it. Most deletion commands operate on small amounts of text. Killing saves the removed text to storage. You can retrieve (referred to as yank) the text at any time.

Characters and Lines

C-d Delete-char. Deletes the character after the cursor.
DEL Delete-backward-char. Deletes the character before the cursor.
C-k Kill-line. Kills to the end of the current line, not including the new-line. Thus, if you are at the beginning of a line, it takes two C-k's to kill the whole line and close up the white space.

Yanking:

Yanking is an other term for retrieving killed text. This is what some systems call pasting. The usual way to move or copy text is to kill it and then to yank it one or more times. You can kill in one buffer, switch to another and yank the text there. To get back previous kills, move around the kill ring (stack). Start with C-y to get the most recent kill, and then use M-y to move to the previous spot in the kill ring by replacing the just-yanked text with the previous kill. Subsequent M-y's move around the ring, each time replacing the yanked text. Stop at the text of interest. Any other command (a motion command, self-insert, anything) breaks the cycling of the kill ring, and the next C-y yanks the most recent kill again.

C-y Yank. Yank last killed text.
M-y Yank-pop. Replace re-inserted killed text with the previous killed text.

Searching:

Emacs has a variety of unusual and extremely powerful search and replace commands. Incremental search is the most important. Begin an incremental search by typing in a character. As each additional character is typed, XEmacs finds and shows where that string of characters is found.

C-s Isearch-forward. Incremental search forward.
C-r Isearch-backward. Incremental search backward.

To stop searching, either hit RET or type any other XEmacs command (which will both stop the search and execute the command). Start the search for the next match by typing another C-s at any point. Reverse the search by typing C-r. Modify the search by using DEL to delete and change the character string.

Query Replace:

Query-replace (bound to M-%) is the most important command for replacing text. This command prompts you for the text to replace, the text to replace it with, and then searches and replaces within the current buffer.

M-% *string* **RET** *newstring* **RET**

Query-replace is interactive: at each match, and will prompt for decision what to do. The following options are available:

SPC	Replace the occurrence with *newstring*.
DEL	Skip the next occurrence without replacing this one.
RET	Terminate query-replace without performing this replacement.
ESC	Same as RET.
. (Period)	Perform this replacement but then terminate the query-replace.
!	Perform this replacement and all the rest in the buffer without asking.

.Emacs file:

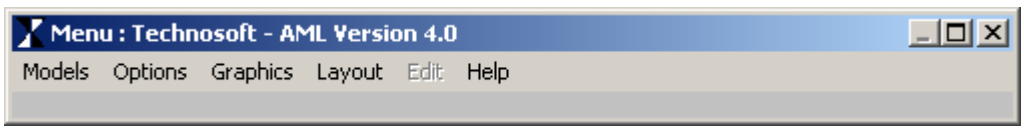
As XEmacs starts, it initializes by reading the ".emacs" file. XEmacs loads any customizations, such as macros, found in the file. The .emacs file is written in the XEmacs Lisp language, which is not to be confused with the AML.

References:

This section contains material taken from the Emacs tutorial (Copyright (c) 1985 Free Software Foundation Richard M. Stallman. GNU Emacs Manual. Cambridge, MA: Free Software Foundation. The complete text is also available on the WWW).

2.3.3 Starting the AML user Interface:

To start the AML user interface, type *(aml)* at the command prompt. The XEmacs environment also has a button (shown at the right) that performs the same command. In addition to displaying the AML user interface, this button will also minimize the XEmacs environment.



This function will bring up the AML menu header bar (shown above) which allows easy access to AML models, layouts and other options when interacting with a model.

2.3.4 Ending an AML session:

The user may end the AML session by two methods:

1. The (*quit)* function closes the complete AML application process including the command prompt and the AML user interface. The editor remains active (without an active AML command prompt buffer).

2. If the AML menu header bar is displayed, press "Models" | "Exit AML", and then confirm the exit. Note that this is the preferred mechanism of exiting AML if the AML user interface has been started.

Overall Model Development Procedure

The overall procedure in model development involves the following steps:

- start the AML application,
- edit file(s) to make class definitions etc.,
- load the files into AML's memory
- start the AML user interface,
- choose a layout from the "Layout" pull-down menu on the menu header bar,
- instantiate the classes using *create-model* or *add-object*
- and inspect/edit/manipulate the results in the AML environment.

Exercise 1

After a close review of the class definitions and AML constructs described in the previous section, develop the class definition for the missile geometry class only using *coordinate-system-class*. Create a class called missile-geometry-class inheriting from *object*. The class should not have any properties. It will have four instances of a *coordinate-system-class* class. Specify the *origin* property of each instance to position each coordinate system instance along the x-axis with a distance of 0.0, 1.0, 9.0, and 17.0 from the *missile-coordinate-system*. Figure 6 shows the instance diagram for the desired hierarchy. The suggested names for the class/subobjects are given first and the class types are given in square brackets [ ].

Instance Diagram for Exercise 1

```
missile-geometry-class [object]
   missile-coordinate-system [coordinate-system-class]
        origin
   nose-coordinate-system [coordinate-system-class]
        origin
   mid-body-coordinate-system [coordinate-system-class]
        origin
   aft-body-coordinate-system [coordinate-system-class]
        origin
```

Figure 6

Follow these steps to create the class and instantiate the object:

- Start the AML application by double clicking on the AML icon,
- Open a new file for AML source code editing,
- Insert the line *(in-package :aml)* at the beginning of the AML source code file (the instructor will explain this after the exercise),
- Define the class as specified above in the new file,
- Save the file to the hard disk,
- Load the file into AML memory so that AML knows the definition of a *missile-geometry-class*,
- Ensure that you have the appropriate AML inspection interface forms displayed (Ex. Menu Form | Layout | AML Main Modeling Form),
- Create an instance of a *missile-geometry-class* using *create-model* at the command prompt,
- Verify that the model tree is current and shows the instance of a *missile-geometry-class* just created,
- Inspect, draw, and modify the instance just created to show some changes in the geometry using the AML browsing interface.

Exercise 1 Solution

```
(in-package :aml)

(define-class missile-geometry-class
  :inherit-from (object)
  :properties (

               )
  :subobjects (
               (missile-coordinate-system :class 'coordinate-system-class
                origin (list 0.0 0.0 0.0)
                )

               (nose-coordinate-system :class 'coordinate-system-class
                origin (list 1.0 0.0 0.0)
                )

               (mid-body-coordinate-system :class 'coordinate-system-class
                origin (list 9.0 0.0 0.0)
                )

               (aft-body-coordinate-system :class 'coordinate-system-class
                origin (list 17.0 0.0 0.0)
                )
               )
  )
```

Code Explanation

The first line of any AML source code file must be *(in-package :aml).* This function tells the compiler to treat the following code as classes/methods/functions organized under a certain bundle of specialized AML functionality. An in-depth discussion of *packages* is beyond the scope of this manual and can be covered in the advanced training.

As specified with a standard class definition, the *missile-geometry-class* inherits from *object*. Four coordinate system subobjects are defined with unique names. Each of the subobjects is an instance of a *coordinate-system-class* with the *origin* property overwritten from the innate definition of *'(0 0 0).*

Notice that all classes in AML must inherit from some other class. These classes can be standard classes that are innate in the AML class library, or pre-defined user-defined classes. The user may notice that some classes have a *–object* or a *–class* suffix. As AML evolves, new classes are being added with a *–class* suffix. The developer is encouraged to use a *–class* suffix when defining his/her own classes.

2.4 The Missile Geometry Class with Coordinate Systems and Components

Figure 7 shows the missile geometry with three components: a conical nose, a cylindrical mid-body, and a truncated conical aft-body. The next step in creating the missile geometry is to use some of the innate geometric primitives in AML to represent the missile's thin-shelled body components.

Missile Nose (Conical), Mid-Body, and Aft-Body

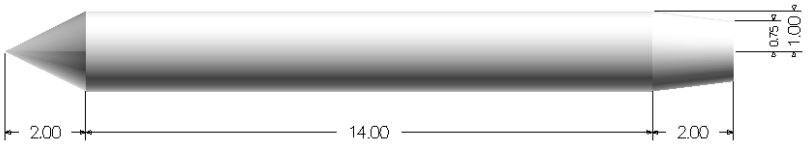


Figure 7

Note: The missile body's radius is 1.0 and the aft body's exit radius is 0.75.

New AML Constructs

The following AML concepts and constructs are used in this section:

- *position-object*
- *graphic-object*
- *open-cone-object*
- *open-cylinder-object*
- *open-truncated-cone-object*
- orientation functions
- keywords in constructs
- *create-model* (using keywords)
- AML documentation

✓ The following classes (*graphic-object and position-object*) are base component classes which are inherited into classes that a user can display within a "display canvas". Each has its own specific purpose and provides certain properties that a user can manipulate in a subclass such as changing an object's color, or the way it is rendered.

POSITION-OBJECT [Class]

The *position-object* provides the ability for objects to be oriented in space. All geometry is created with its own local origin at the absolute (global) origin, *'(0.0 0.0 0.0)*, by default.

Properties:

orientation	The list of orientation commands used to position the object (see section on orientation)

GRAPHIC-OBJECT [Class]

All objects that have geometry and graphics associated with them inherit from *graphic-object*.

Inherit-From:

position-object

Properties:

color	The value may be a symbol, a string, or a list of three numbers between 0 and 1 specifying the red, green, and blue components of the color. Default is *"white"*. Other examples are *'white*, or *'(0.2 1 0.5)*.
display?	When t, an instance of this class will be capable of being drawn. When nil, an instance cannot be displayed. Default t.
render	*'boundary* for wireframe graphics, *'shaded* for a shaded representation, and *'facet* to connect the surface facets with lines. Other options are available such as *'boundary-shaded*, and *'facet-shaded*.
line-width	The width of the lines used to draw the object.
line-type	The style of the lines used to draw the object.

OPEN-CONE-OBJECT [Class]

An *open-cone-object* is defined as an open ended hollow cone.

Inherit-From:

graphic-object

Properties:

diameter	The diameter of the end circle. Default 0.5.
height	The height is defined parallel to the z-axis. Default 2.0.

OPEN-CYLINDER-OBJECT [Class]

An *open-cylinder-object* is a open ended hollow cylinder.

Inherit-From:

graphic-object

Properties:

diameter	The diameter of the cylinder. Default 1.0.
height	The height is defined parallel to the z-axis. Default 2.0.

Example:

```
(define-class OPEN-COLUMN-CLASS
:inherit-from (open-cylinder-object)
:properties (
        diameter         40.0
        height           150.0

        color            'white
        )
    )
```

See Also:

```
pipe-object
```

OPEN-TRUNCATED-CONE-OBJECT [Class]

An *open-truncated-cone-object* is defined as an open ended hollow frustum of a cone.

Inherit-From:

graphic-object

Properties:

start-diameter	The diameter of the start face of the truncated cone located in the negative z-direction space. Default is 1.0.
end-diameter	The diameter of the end face of the truncated cone located in the positive z-direction space. Default is 1.0.
height	The height of the truncated cone. Default is 2.0.

Orientation Functions

All objects which inherit from *position-object* can be oriented in model space. An object's orientation can consist of any combination of translations and rotations. This orientation may be "built-into" the object through its class definition, or applied to the object after instantiation (creation) through the orientation form in the AML graphical user interface.

The format of the orientation property is as follows:

```
orientation   (list   (operation-1 args)
                      (operation-2 args)
                      ...
                      )
```

✓ The formula for *orientation* cannot be defined as a quoted list (') because it is not evaluated in the same manner as other properties.

Consider the example of an *open-cylinder-object* translated and rotated away from its original position.

```
(define-class example-cyl-class
    :inherit-from (open-cylinder-object)
    :properties (
        orientation (list   (translate '(5.0 0.0 0.0))
                            (rotate 45.0 '(0.0 1.0 0.0))
                    )
    )
)
```

The *orientation* property specifies that the cylinder should first be translated a distance of 5.0 along the x-axis (of the global coordinate frame). After that, it is rotated by 45.0 degrees about the y-axis, or the vector (0 1 0) of the global coordinate frame. The orientation operations are built into the object, therefore any instance of this object will be immediately transformed to the new orientation on creation. Note that the order of the operations in orientation is important. Reversal of the operations,

```
        orientation (list   (rotate 45.0 '(0.0 1.0 0.0))
                            (translate '(5.0 0.0 0.0))
                    )
```

will result in a different final orientation. The orientation functions can contain expressions which will be evaluated when the object is created. Thus, to move the *open-cylinder-object* along the x-axis by a distance equal to twice its *diameter* and then rotate it, the following orientation can be used using *the* referencing (see next section for explanation):

```
            orientation (list   (translate (list (* 2.0 ^diameter) 0.0 0.0)
                                (rotate 45.0 '(0.0 1.0 0.0))
                        )
```

Example:

```
    (define-class example-cyl-object
    :inherit-from (open-cylinder-object)
    :properties (
            orientation (list   (translate '(5.0 0.0 0.0))
                                (rotate 45.0 '(0.0 1.0 0.0))
                                (translate '(2.0 1.0 0.0))
                                (rotate 25.0 '(0 0 1))
                        )
            )
        )
```

Keywords

Within both methods and functions, there are arguments that are not required at the time of the functions execution. They simply have a default condition which they evaluate to in the event that they are not provided by the user/developer. These keywords help to maintain the ease of programming in AML, without limiting its overall functionality. When called in a method or function, a colon : is placed before the keyword, the keyword is given and then the keyword's argument is given. In the following example the orientation command *translate* has a keyword called *distance* that is called with a value of 3.158.

Example:

```
    (define-class example-cyl-class
    :inherit-from (open-cylinder-object)
    :properties (
            orientation (list   (translate '(1.0 0.0 0.0) :distance 3.158))
            )
        )
```

CREATE-MODEL [Function]

As has been previously shown, *create-model* allows the user to create a new model based on the specified class. Now that the topic of keywords has been covered, this function is revisited to demonstrate further capability available in *create-model*.

Format:

(CREATE-MODEL name [:class class-name])

Arguments:

name	A symbol describing the user defined name for the model being created. If the class argument is not supplied, the name must be a valid class name.
:class	A symbol describing the class name. The default value for class is the name. A valid class name is required.

Examples:

```
AML> (create-model 'model-1 :class 'object)
#<OBJECT @ #x2224280a>
```

This creates a model named *model-1* of class *object*.

AML Documentation

AML Reference Manual:

TechnoSoft encourages AML developers to use the AML Reference Manual for questions about AML syntax or class structure. For quick reference, the AML Reference Manual contains:

- a table of contents which separates the entire manual with respect to content,
- an index which separates AML constructs into
  - General Constructs,
  - Classes,
  - Functions,
  - And Methods.

Runtime Documentation:

AML has several methods of providing help while at an AML command prompt. In going through this training manual, the AML developer may find the AML runtime documentation functionality useful. TechnoSoft suggests using the *apropos* and *describe* at the command prompt. *apropos* finds all text in the current AML session that matches a given symbol. *describe* prints information about a given object.

```
AML> (describe 'create-model)
CREATE-MODEL is a SYMBOL.
  It is unbound.
  It is EXTERNAL in the TechnoSoft package and accessible in the
Adaptive-Modeling-Language, CHISELS, and VirtualGeometryLayer packages.
  Its function binding is #<Function CREATE-MODEL>
    The function takes arguments (NAME &KEY CLASS DELETABLE? INIT-FORM)
AML> (create-model 'missile-geometry-class)
#<MISSILE-GEOMETRY-CLASS @ #x21ec4322>
AML> (expand (the))
NIL
AML> (describe (the))

DEFINE-CLASS PROPERTIES:
------------------------
```

```
DEFINE-CLASS SUBOBJECTS:
------------------------
 MISSILE-COORDINATE-SYSTEM  #<COORDINATE-SYSTEM-CLASS>
 NOSE-COORDINATE-SYSTEM     #<COORDINATE-SYSTEM-CLASS>
 MID-BODY-COORDINATE-SYSTEM #<COORDINATE-SYSTEM-CLASS>
 AFT-BODY-COORDINATE-SYSTEM #<COORDINATE-SYSTEM-CLASS>
NIL
AML> (apropos 'missile)
MISSILE
MISSILE-GEOMETRY-CLASS
MISSILE-COORDINATE-SYSTEM
```

✓ The trainee is encouraged to look in the AML reference manual to view the documentation on the various orientation commands.

Exercise 2

After a close review of the class definitions and AML constructs described in the previous section, augment the class definition for the missile geometry class using instances of *coordinate-system-class*, *open-cone-object*, *open-cylinder-object*, and *open-truncated-cone-object* as subobjects in the *missile-geometry-class*. Note that Figure 8 shows properties that will need to be changed in the various objects to make the missile geometry given in Figure 7. These objects have more properties than shown here, but these are only the ones you will need to change.

Instance Diagram for Exercise 2

```
missile-geometry-class [object]
   missile-coordinate-system [coordinate-system-class]
   nose-coordinate-system [coordinate-system-class]
        origin
   nose [open-cone-object]
        diameter
        height
        orientation
   mid-body-coordinate-system [coordinate-system-class]
        origin
   mid-body [open-cylinder-object]
        diameter
        height
        orientation
   aft-body-coordinate-system [coordinate-system-class]
        origin
   aft-body [open-truncated-cone-object]
        start-diameter
        end-diameter
        height
        orientation
```

Figure 8

Exercise 2 Solution

```
(in-package :aml)

(define-class missile-geometry-class
  :inherit-from (object)
  :properties (

               )
  :subobjects (
               (missile-coordinate-system :class 'coordinate-system-class
                origin (list 0.0 0.0 0.0)
                )

               (nose-coordinate-system :class 'coordinate-system-class
                origin (list 1.0 0.0 0.0)
                )

               (nose :class 'open-cone-object
                 height              2.0
                 diameter            2.0
                 orientation         (list
                                      (rotate -90 '(0 1 0))
                                      (translate (list 1.0 0.0 0.0))
                                      )
                )

               (mid-body-coordinate-system :class 'coordinate-system-class
                origin (list 9.0 0.0 0.0)
                )

               (mid-body :class 'open-cylinder-object
                 diameter               2.0
                 height                 14.0
                 orientation            (list
                                         (rotate 90 '(0 1 0))
                                         (translate (list 9.0 0.0 0.0))
                                          )
                )

               (aft-body-coordinate-system :class 'coordinate-system-class
                 origin                    (list 17.0 0.0 0.0)
                )

               (aft-body :class 'open-truncated-cone-object
                 start-diameter         2.0
                 end-diameter           1.5
                 height                 2.0
                 orientation            (list
                                           (rotate 90 '(0 1 0))
                                           (translate (list 17.0 0.0 0.0))
                                           )
                 )
               )
  )
```

Code Explanation

Note that the *missile-geometry-class* now has several subobjects that have hard-coded properties. To change the geometric configuration of the missile, the user would have to individually inspect each object and change the values, or the developer would have to create a user interface link to each property in the various locations through the model. Also, most of the properties rely on each other and should be parametrically linked so the user/developer does not have to manage all of the locations where the information is used. One instance of this can be seen in the missile's radius controls the radii of the nose, mid-body, and the start diameter of the aft-body.

2.5 The Missile Geometry Class with Parametrically Designed Components

The next step in creating the missile geometry is to augment the missile's design by creating parametric relations to certain "top-level" properties from which the components will derive their necessary information. The coordinate systems will also drive the placement of the components to further promote the parametric design.

New AML Constructs

The following AML concepts and constructs are used in this section:

- Data Model Concept (Common Computational Model)
- *the* referencing
- *the* shortcut… "!"
- *superior*
- *superior* shortcut… "^"
- *reference-coordinate-system* property in *position-object*

Organizing Data in a Central Location - Data Model Concept

AML provides a Knowledge Based Engineering (KBE) system for modeling and capturing knowledge from different engineering domain disciplines. Products, Methods, and Processes are represented in a common computational object hierarchy model facilitating the reuse of the product as well process knowledge. It captures and organizes the vital engineering knowledge and processes within a unified distributed object-oriented part model enabling the seamless integration of engineering tools to automate the entire engineering cycle from conceptual design to production. With this in mind, it is a common practice to organize common properties and objects of an AML model or class at one "place" within the model so other properties and objects can access them easily. The properties should contain knowledge/data about the particular class such that all of the subobjects and sibling properties can access the common information. In the case of the *missile-geometry-class*, the *nose*, *mid-body* and *aft-body* each need the *missile-general-body-radius* property. This concept can be expanded to such areas as cost analysis relying on the geometric and material properties of an object, the stress analysis of an object relying on the geometric, material, and load properties within a model, and several other examples.

THE Referencing

the is an AML construct which interrogates an object for its properties and subobjects. A developer can use *the* referencing to "get" properties, objects, or values of properties from other places in the instance hierarchy and to "describe" where they are located in the the instance hierarchy. For example, consider an instance hierarchy of an airplane shown below in an "instance diagram" where capitalized words represent objects and lower case words represent properties of those objects:

```
AIRPLANE                        [Level 1]
    maximum-speed               [Level 2]
    wing-span                   [Level 2]
    number-of-engines           [Level 2]
    WINGS                       [Level 2]
        WING-0001               [Level 3]
            span                [Level 4]
        WING-0002               [Level 3]
            span                [Level 4]
            RIBS                [Level 4]
                RIB-0001        [Level 5]
                    length      [Level 6]
                    width       [Level 6]
    FUSELAGE                    [Level 2]
        length                  [Level 3]
        radius                  [Level 3]
    TAIL-SECTION                [Level 2]
        ELEVATOR                [Level 3]
            deflection-angle    [Level 4]
```

Assuming this hierarchy, the developer can navigate through the instances and properties via *the* referencing. Conceptually, *the* referencing has start point and an end point. The end point is the name of the target object or property. The start point varies depending on where the *the* reference is coded. From the AML prompt, *the* referencing "starts" from the root object in the model tree. When writing code in files, *the* referencing starts from the current "level" in the tree. From *methods* (described in the "Optional Topics" section of this manual), *the* referencing starts from the instance the method is called on.

Assume the developer starts at the *AIRPLANE* level, and wants to obtain the value stored in *span* from one of the *WINGS*. The following call is made from the command prompt (assuming the current model is the *AIRPLANE*:

```
AML> (the airplane wings wing-0001 span)
40.21
```

Assume the developer starts at the *AIRPLANE* level, and wants to obtain the object stored in *WING-0001*.

```
AML> (the airplane wings wing-0001)
#<WING-CLASS #x214H658>
```

This call returns an the instance of a *wing-class* which is an object data type versus the float data type returned in the previous example.

A *the* reference will continue to look up the tree for an object or property with the name of the target. It will go up to the superior superior of its own level, and then look at that object's children or properties to find the target. If one is found, the value of that property is returned. Otherwise, the a similar operation is performed by going up another "superior" until it reaches the root of the tree. If it does not find the target, *the* returns an error value.

Format:

(the name-1 name-2 ... name-n)

Examples:

```
(define-class MATERIAL-CLASS
    :inherit-from (object)
    :properties (
        material        'wood
        density         0.0
        )
    )

(define-class EXAMPLE-SUBOBJECT-CLASS
    :inherit-from (material-class)
    :properties (
        id              "OU812"
        part-number     2.0
        material        'steel
        )
    )

(define-class EXAMPLE-MODEL-CLASS
    :inherit-from (object)
    :properties (
        model-name      "Model For AML Basic Training"
        id              "OU812B4"
        )
    :subobjects (
        (sub-1 :class 'example-subobject-class
            part-number     3.0
            )
        (sub-2 :class 'example-subobject-class
            id              "ZZ184"
            material        'aluminum
            )
        )
    )

AML> (create-model 'example-model-class)
#<EXAMPLE-MODEL-CLASS #x214H658>        ➔returns the object … an instance
AML> (the)
#<EXAMPLE-MODEL-CLASS #x214H658>        ➔returns the object … an instance
AML> (the example-model-class)
#<EXAMPLE-MODEL-CLASS #x214H658>        ➔returns the object … an instance
AML> (the example-model-class id)
"OU812B4"                   ➔returns the value of the property … a string
AML> (the example-model-class sub-1 id)
"OU812"                     ➔returns the value of the property … a string
AML> (the example-model-class sub-1)
#<EXAMPLE-SUBOBJECT-CLASS @ #x21c1fdea> ➔returns the object … an instance
AML> (the example-model-class sub-1 material)
STEEL                       ➔returns the value of the property … a symbol
AML> (the example-model-class sub-2 material)
ALUMINUM                    ➔returns the value of the property … a symbol
```

See Also:

Default

! Shortcut

To produce more readable code, and to reduce typing, AML has a shortcut for *the* specified with an exclamation point (*!*). The best way to introduce this shortcut is by example, but as a general rule *!xyz* is exactly the same as *(the xyz)*. Note that no spaces are allowed in this shortcut between the exclamation point and the *xyz*.

Examples:

Assuming the same model is at the root *(root-object)*, or *current-model*

```
AML> (the example-model-class)
#<EXAMPLE-MODEL-CLASS #x214H658>
AML> !example-model-class
#<EXAMPLE-MODEL-CLASS #x214H658>
AML> (the sub1 part-number)
3
AML> !sub1 part-number
#<EXAMPLE-SUBOBJECT-CLASS #x692H003>
```

Notice how the *part-number* argument given in the last example is ignored because spaces are not allowed with the exclamation point shortcut.

SUPERIOR

The *superior* property makes *the* referencing more efficient and helps produce more readable code. Essentially, each *superior* takes a *the* reference up one level. Note that this still is a *the* reference and will continue to search up the tree if it does not find the target one level above itself.

Example:

Assume the following class definitions:

```
(define-class EXAMPLE-SUBOBJECT-CLASS
:inherit-from (object)
:properties (
        id              “OU812”
        part-number     2.0
        district-code   nil
    )
)
(define-class EXAMPLE-MODEL-CLASS
:inherit-from (object)
:properties (
        model-name              "Model For AML Basic Training "
        code                    45242
        )
:subobjects (
        (sub-1 :class ‘example-subobject-class
            id                  (the superior superior sub-2 id)
            part-number         3.0
```

```
            district-code       (the superior superior code)
        )

        (sub-2 :class 'example-subobject-class
            id                  "ZZ184"
            district-code       (the superior superior code)
        )
    )
)
```

The *district-code* of *sub-1* and *sub-2* get their values from their parents parent; *example-model-class.* This is just one illustration that shows automation of data flow in AML with the use of *superior superior*.

The *id* of *sub-2* gets its value from redefinition at instantiation as a subobject to *exapmle-model-class.* The *id* of *sub-1* gets its value by employing a the-reference to the *id* of *sub-2*.

There is also a shortcut for the *superior* in AML specified with a ^ (read as "the superior). The shortcut is defined as:

^*xyz = (the superior xyz)* and

^^*xyz = (the superior superior xyz)* and so on …

Examples:

```
(define-class EXAMPLE-MODEL-CLASS
:inherit-from (object)
:properties (
        category                "example models"
        code                    45242
        )
:subobjects (
        (sub-1 :class 'example-subobject-class
            id                  (the superior superior sub-2 id)
            part-number         3.0
            district-code       ^^code
        )

        (sub-2 :class 'example-subobject-class
            id                  "ZZ184"
            district-code       ^^code
        )

    )
)
```

Note: Notice that the formula of *sub-1 id* could not change due to the nature of its the-reference (it would need spaces in between the *sub-2* and *id* which is not allowed).

POSITION-OBJECT (expanded to show *reference-coordinate-system*) [Class]

The *position-object* provides the ability for objects to be oriented in space. The *position-object* definition is expanded here to show the *reference-coordinate-system* property which allows an object to orient itself with respect to an instance of a *coordinate-system-class*.

Properties:

orientation — The list of orientation commands used to position the object (see section on orientation)

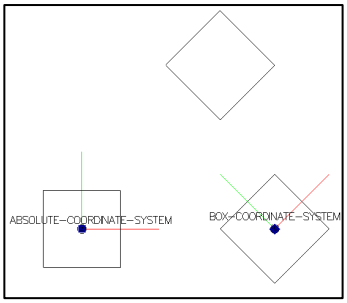
reference-coordinate-system

Specifies the *coordinate-system-class* object which will be the reference frame for this object. The *reference-coordinate-system* property must point to an object of class *coordinate-system-class*. A value of *nil* indicates that the global reference frame should be used. If the *reference-coordinate-system* refers to any object which does not inherit from *coordinate-system-class*, the global frame will be used. By default, the reference frame is global. (Defaults to *nil*)

Examples:

The following example uses a *box-object* to demonstrate the use of the *reference-coordinate-system* property. Note that in AML, a box's local origin is located at its centroid.

```
(define-class reference-coordinate-system-example-class
  :inherit-from (object)
  :properties (

               )
  :subobjects (
               (absolute-coordinate-system :class 'coordinate-system-class
                )
               (box-without-orientation :class 'box-object
                )
               (box-coordinate-system :class 'coordinate-system-class
                origin (list 2.5 0.0 0.0)
                vector-i (list 1.0 1.0 0.0)
                vector-j (list -1.0 1.0 0.0)
                )
               (box-with-ref-coord-sys :class 'box-object
                reference-coordinate-system ^^box-coordinate-system
                )
               (box-with-orientation :class 'box-object
                reference-coordinate-system ^^box-coordinate-system
                orientation (list
                             (translate (list 1.0 2.0 0.0))
                             )
                )
               )
  )
```



Exercise 3

After a close review of the class definitions and AML constructs described in the previous section, use *the* referencing to augment the class *missile-geometry-class* to contain the following properties with their given values/formulas:

- missile-general-body-radius → 1.0
- missile-nose-length → 2.0
- missile-nose-radius → parameterize to missile-general-body-radius
- missile-mid-body-length → 14.0
- missile-mid-body-radius → parameterize to missile-general-body-radius
- missile-aft-body-length → 2.0
- missile-aft-body-start-radius → parameterize to missile-general-body-radius
- missile-aft-body-end-radius → 0.75

Use the *reference-coordinate-system* property of each class that inherits from *position-object* to parameterize the missile's orientation. The *missile-coordinate-system* is the driving orientation device for the entire missile geometry. The *aft-body-coordinate-system* should reference the *mid-body-coordinate-system* which references the *nose-coordinate-system* which references the *missile-coordinate-system.* Each missile geometry component should also reference their respectively named coordinate-system. Figure 9 shows the instance diagram for the desired hierarchy.

Instance Diagram for Exercise 3

```
missile-geometry-class [object]
   missile-general-body-radius
   missile-nose-length
   missile-nose-radius
   missile-mid-body-length
   missile-mid-body-radius
   missile-aft-body-length
   missile-aft-body-start-radius
   missile-aft-body-end-radius
   missile-coordinate-system [coordinate-system-class]
   nose-coordinate-system [coordinate-system-class]
        origin
        reference-coordinate-system
   nose [open-cone-object]
        diameter
        height
        orientation
        reference-coordinate-system
   mid-body-coordinate-system [coordinate-system-class]
        origin
        reference-coordinate-system
   mid-body [open-cylinder-object]
        diameter
        height
        orientation
        reference-coordinate-system
   aft-body-coordinate-system [coordinate-system-class]
        origin
        reference-coordinate-system
   aft-body [open-truncated-cone-object]
        start-diameter
        end-diameter
        height
        orientation
        reference-coordinate-system
```

Figure 9

Exercise 3 Solution

```
(in-package :aml)

(define-class missile-geometry-class
  :inherit-from (object)
  :properties (
           missile-general-body-radius      1.0

           missile-nose-length              2.0
           missile-nose-radius              ^missile-general-body-radius

           missile-mid-body-length          14.0
           missile-mid-body-radius          ^missile-general-body-radius

           missile-aft-body-length          2.0
           missile-aft-body-start-radius    ^missile-general-body-radius
           missile-aft-body-end-radius      0.75
           )
  :subobjects (
              (missile-coordinate-system :class 'coordinate-system-class
               origin (list 0.0 0.0 0.0)
               )

              (nose-coordinate-system :class 'coordinate-system-class
                origin   (list (* 0.5 ^^missile-nose-length) 0.0 0.0)
                reference-coordinate-system ^^missile-coordinate-system
               )

              (nose :class 'open-cone-object
                height             ^^missile-nose-length
                diameter           (* ^^missile-nose-radius 2.0)
                orientation        (list
                                    (rotate -90 '(0 1 0))
                                    )
                reference-coordinate-system ^^nose-coordinate-system
                )

              (mid-body-coordinate-system :class 'coordinate-system-class
               origin       (list
                                  (+ (/ ^^missile-nose-length 2.0)
                                     (/ ^^missile-mid-body-length 2.0)
                                     )
                                  0.0
                                  0.0)
               reference-coordinate-system ^^nose-coordinate-system
               )

              (mid-body :class 'open-cylinder-object
                diameter     (* ^^missile-mid-body-radius 2.0)
                height       ^^missile-mid-body-length
                orientation  (list
                                  (rotate 90 '(0 1 0))
                                  )
                reference-coordinate-system ^^mid-body-coordinate-system
                )

              (aft-body-coordinate-system :class 'coordinate-system-class
                origin       (list
                                 (+ (/ ^^missile-mid-body-length 2.0)
                                    (/ ^^missile-aft-body-length 2.0)
                                    )
```

```
                                0.0
                                0.0)
                  reference-coordinate-system ^^mid-body-coordinate-system
                 )

                (aft-body :class 'open-truncated-cone-object
                  start-diameter (* ^^missile-aft-body-start-radius 2.0)
                  end-diameter   (* ^^missile-aft-body-end-radius 2.0)
                  height         ^^missile-aft-body-length
                  orientation    (list
                                       (rotate 90 '(0 1 0))
                                       )
                  reference-coordinate-system ^^aft-body-coordinate-system
                  )
                )
    )
```

Code Explanation

The *missile-geometry-class* now has several properties that allow a user to control the geometric parameters from one location. The subobjects have properties that refer to these top-level properties via *the* references. These *the* references create dependencies on their respective target properties or objects. AML manages these dependencies automatically without the need for any further user/developer interactions. When the top-level properties change, the respective properties in the subobjects will become "unbound", meaning that they are no longer valid. Upon need (also known as being demanded), the property's formula will be recalculated thus making the property "bound". This is known as demand-driven calculation. Properties and objects in AML are only calculated or instantiated when demanded.

✓ Notice how the property missile-general-body-radius controls the radii of the nose, mid-body, and the start diameter of the aft-body. This demonstrates the power of a common computational model. All requirements and data can be obtained and interfaced from one common location. All parameters that may need this information can obtain it from one consistently located place and all dependencies are automatically managed internally.

2.6 The Missile Geometry Class with Optional Nose Type

Figure 10 shows the missile geometry with three components: a spherical nose, a cylindrical mid-body, and a truncated conical aft-body. The next step in creating the missile geometry is to augment the missile's design components with the option of having a spherical shaped nose. This introduces the AML syntax for using conditions and enabling various options to the user.

Missile Nose (Spherical), Mid-Body, and Aft-Body

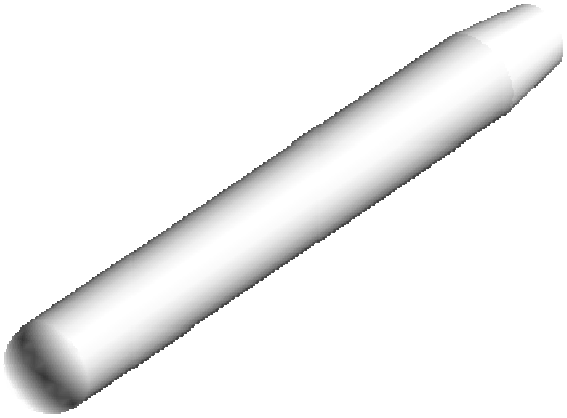


Figure 10

New AML Constructs

The following AML concepts and constructs are used in this section:

- *case*
- *if*
- *default*
- *solid-object*
- *sphere-object*
- comments

The *case* statement compares a test-key to a number of keys and evaluates the expressions that are included with the first matching key.

Format:

(CASE test-key (key1 expressions) [(key2 expressions)]...[(keyN expressions)]

Arguments:

test-key	Any expression that will evaluate to a result that gets compared against all of the keys. **This value must be a symbol or a number.**
key	A symbol or list of symbols or numbers that will be compared to test-key. If the value is a list the key-test is compared to each of the keys in the list. A key of t may be included as the last key to operate as a default that will match in all instances.
expressions	Any number of expressions to be evaluated if the key matches the test-key.

Examples:

```
AML> (case 3
        (1 "Value: One")
        (2 "Value: Two")
        (3 "Value: Three")
        (t "Value: Unknown"))
"Value: Three"
AML> (case 5
        (1 "Value: One")
        (2 "Value: Two")
        (3 "Value: Three")
        (t "Value: Unknown"))
"Value: Unknown"
AML> (case 5
        (1 "Value: One")
        (2 "Value: Two")
        (3 "Value: Three")
    )
NIL
AML> (case nil
        (john   (+ 2 3))
        (steve  (+ 6 3))
        (mark   (+ 8 3))
        (t      "Value: Unknown")
    )
"Value: Unknown"
AML> (case 'cone
        (cone   'open-cone-object)
        (sphere 'sphere-object)
        (t      'open-cone-object))
OPEN-CONE-OBJECT
AML> (define-class conditional-subobject-class
    :inherit-from (object)
    :properties (
```

```
                subobject-class-type  'box
                )
        :subobjects (
                (sub-1 :class (case !subobject-class-type
                                (box    'box-object)
                                (cone   'open-cone-object)
                                (sphere 'sphere-object)
                                (t      'open-cone-object))
                        )
                )
        )
```

See Also:

Cond

IF **[Function]**

If the *test-expression* is not nil, the *true-expression* clause is evaluated and if the *test-expression* is nil, the *false-expression* clause is evaluated. The *false-expression* clause is optional and will return nil if not included. Because it only allows a single expression for the *true-expression* and *false-expression* clauses, it is sometimes necessary to use a *progn* statement, which treats many expressions as a single function, for one of these clauses.

✓ Use *nil* to represent a false value in AML. The only expression or data entity in AML that is false is *nil*; all others are true. This means a number, an object, or a list are all true.

Format:

(IF test-expression true-expression [false-expression])

Arguments:

test-expression	Any expression that will be used to determine whether to evaluate the *true-expression* or *false- expression*.
true-expression	Any single expression that will be evaluated when *test-expression* is not nil.
false-expression	Any optional single expression that will be evaluated when the *test-expression* is nil.

Examples:

```
AML> (if (= 2 3) "This is true." "This is false.")
"This is false."
AML> (if (< 2 3) (+ 3 5) (- 3 5))
8
AML> (if (> 2 3) (+ 3 5))
NIL
AML> (if t (list “abc” 1 2) nil)
(“abc” 1 2)
AML> (if nil (list “abc” 1 2) nil)
NIL
AML> (if (+ 1 2 3) (list “abc” 1 2))
```

```
("abc" 1 2)
```

See Also:

```
and
or
when
=
equal
not
progn
```

DEFAULT [Function]

When specified as the formula for a property, *default* will look up the tree for an object with a property with the same name. If one is found, the value of that property is returned. Otherwise, the specified default-formula is used.

✓ The use of *default* creates a dependency and "climbs" the tree hierarchy similar to a *the* reference.

Format:

(DEFAULT [default-formula])

Arguments:

default-formula

If the search up the tree for a property with the same name is unsuccessful, the formula specified here will be used. If no formula is specified, the default will be a *popup-typein*.

Example:

```
(define-class DEFAULT-TEST-CLASS
:inherit-from (object)
:properties (
    height 15
    width 10
    depth 6
    material 'steel
    )
:subobjects (
    (box1 :class 'box-object
        height (default 13)
        width 9
        density 6
        surface-finish (default '(smooth 0.001))
        )
    (box2 :class 'box-object
        height 11
        depth (default 5)
        material (default 'wood)
        )
    )
)
```

```
AML> (create-model 'default-test-class)
#<DEFAULT-TEST-CLASS #x19A38EC>
AML> (the height)
15
AML> (the box1 height)
15
AML> (the box1 width)
9
AML> (the box1 depth)
6
AML> (the box2 depth)
6
AML> (the box2 material)
STEEL
AML> (the box2 width)
10
AML> (the box1 surface-finish)
(SMOOTH 0.001)
```

✓ The use of *default* is highly encouraged in AML. This can provide many functional advantages in writing and integrating classes. These are explained below:

- With specifically named properties, classes may be integrated together with little or no need to manually enter specific *the* references to similarly named properties "above" the property using *default* in its formula.
- This promotes modular and dynamic design. You can make general classes that "obtain" properties from objects "above" them without specifically creating *the* references.
- The use of long, specific property name is encouraged. *default* takes advantage of this practice to enable less code writing. This can also provide undesired results in properties such as *diameter,* and *height* in such classes as *cylinder-object*¸ *sphere,* and *box-object*.
- As mentioned above, some properties may have the same name within a hierarchy. This may lead to the property located "below" getting the value from "above". This is seen in the *default-test-class* example where *(the box1 depth)* is not specifically written in the code. However, the default formula of *depth* from a *box-object* uses the *default* function. Therefore, it will look "up" the tree for an other property called *depth*.

SOLID-OBJECT [Class]

Solid-object provides three dimensional geometric primitive objects a property to determine if the geometry is solid or a hollow shell.

Inherit-From:

object

Properties:

solid?	When true, the geometry will be a solid. When nil, the geometry will be created as a hollow shell. Changing the property will cause the geometry to update. Default t.

✓ Note the use of a question mark "?" in the *solid?* property. This is a convention in AML modelling that denotes a property typically evaluating to true or false (*t* or *nil*). When creating user-defined classes, this practice is also encouraged.

SPHERE-OBJECT [Class]

The *sphere-object* may be defined as a geometric shell or solid. Setting the *solid?* property to true will create a solid.

Inherit-From:

graphic-object, solid-object

Properties:

diameter	Default 1.0

Example:

```
(define-class BEACH-BALL-CLASS
        :inherit-from (sphere-object)
        :properties (
                    diameter         2.5

                    color            'blue
                    render           'shaded
                    solid?           nil
                    )
    )
```

Comments

Comments in AML are specified with a semicolon ";". Any words, numbers, expressions, or characters after the semicolon will be ignored by the compiler until the beginning of the next line.

Examples:

```
AML> (+ 5 1)  ;you may type after the semicolon
6
AML> (define-class BEACH-BALL-CLASS
        :inherit-from (sphere-object)
        :properties (
                    ;;user defined properties
                    diameter         2.5
                    color            'blue
                    render           'shaded

                    ;;internal properties
                    solid?           nil
                    radius           (/ ^diameter 2)
                    )
    )
```

Exercise 4

After a close review of the class definitions and AML constructs described in the previous section, define two new classes for to be instantiated as nose objects and two new classes for the mid-body and aft-body respectively.

Figure 11 shows the instance diagram for a class called *spherical-nose-class* which inherits from *sphere-object*. Figure 11 also shows the instance diagram for a class called *conical-nose-class* which inherits from *open-cone-object*. Give each of these classes the necessary properties to effectively use *default* when they are instantiated in the *missile-geometry-class*. Augment the class definition for the missile geometry class to use instances of *spherical-nose-class*, and *open-conical-nose-class*. Add a property called *missile-nose-type* that contains a symbol data type for a formula. The symbol can be either '*sphere* or '*cone*. The *nose* subobject's class type will depend on this property's value. In this example, the coordinate system of the *spherical-nose-class* is located at the flat face of the object, not its center.

Some of the properties in the classes can effectively use *default*, especially if they are specifically named. Change these properties so that their formulas will take advantage of the *default* functionality, "look up", to obtain values from similarly named properties when instantiated inside of an other object. Namely, *missile-nose-radius*, and *missile-nose-length* can take advantage of the *default* functionality.

Instance Diagrams for Exercise 4

```
spherical-nose-class [sphere-object]
   missile-nose-radius
   diameter

open-conical-nose-class [open-cone-object]
   missile-nose-length
   missile-nose-radius
   height
   diameter
   orientation

open-cylindrical-body-class [open-cylinder-object]

open-truncated-cone-body-class [open-truncated-cone-object]

missile-geometry-class [object]
   missile-general-body-radius
   missile-nose-length
   missile-nose-radius
   missile-nose-type
   missile-mid-body-length
   missile-mid-body-radius
   missile-aft-body-length
   missile-aft-body-start-radius
   missile-aft-body-end-radius
   missile-coordinate-system [coordinate-system-class]
   nose-coordinate-system [coordinate-system-class]
        origin
        reference-coordinate-system
   nose [spherical-nose-class or open-conical-nose-class]
        reference-coordinate-system
   mid-body-coordinate-system [coordinate-system-class]
        origin
        reference-coordinate-system
```

```
mid-body [open-cylindrical-body-class]
     diameter
     height
     orientation
     reference-coordinate-system
aft-body-coordinate-system [coordinate-system-class]
     origin
     reference-coordinate-system
aft-body [open-truncated-cone-body-class]
     start-diameter
     end-diameter
     height
     orientation
     reference-coordinate-system
```

Figure 11

Exercise 4 Solution

```
(in-package :aml)

(define-class spherical-nose-class
  :inherit-from (sphere-object)
  :properties (
               ;;user defined properties
               missile-nose-radius (default 1.0)

               ;;internal properties
               solid?              nil
               diameter            (* ^missile-nose-radius 2.0)
               )
  )

(define-class open-conical-nose-class
  :inherit-from (open-cone-object)
  :properties (
               ;;user defined properties
               missile-nose-length (default 2.0)
               missile-nose-radius (default 1.0)

               ;;internal properties
               height              ^missile-nose-length
               diameter            (* ^missile-nose-radius 2.0)
               orientation         (list
                                    (rotate -90 '(0 1 0))
                                    )
               )
  )

(define-class open-cylindrical-body-class
  :inherit-from (open-cylinder-object)
  :properties (

               )
  )

(define-class open-truncated-cone-body-class
  :inherit-from (open-truncated-cone-object)
  :properties (

               )
  )

(define-class missile-geometry-class
  :inherit-from (object)
  :properties (
        ;;user defined properties
        missile-general-body-radius     (default 1.0)

        missile-nose-length   (default 2.0)
        missile-nose-radius   (default ^missile-general-body-radius)
        missile-nose-type     (default 'sphere)
                              ;; options are 'sphere or 'cone

        missile-mid-body-length (default 14.0)
        missile-mid-body-radius (default ^missile-general-body-radius)

        missile-aft-body-length       (default 2.0)
        missile-aft-body-start-radius (default ^missile-general-body-radius)
        missile-aft-body-end-radius   (default 0.75)
```

```
            )
:subobjects (
            (missile-coordinate-system :class 'coordinate-system-class
             origin (list 0.0 0.0 0.0)
             )

            (nose-coordinate-system :class 'coordinate-system-class
              origin  (if (equal ^^missile-nose-type 'cone)
                          (list (* 0.5 ^^missile-nose-length) 0.0 0.0)
                          (list ^^missile-nose-radius 0.0 0.0)
                          )
              reference-coordinate-system ^^missile-coordinate-system
             )

            (nose :class (case !missile-nose-type
                           (sphere   'spherical-nose-class)
                           (cone     'open-conical-nose-class)
                           (t        'spherical-nose-class)
                           )
              reference-coordinate-system ^^nose-coordinate-system
             )

            (mid-body-coordinate-system :class 'coordinate-system-class
             origin  (if (equal ^^missile-nose-type 'cone)
                                (list (+ (/ ^^missile-nose-length 2.0)
                                         (/ ^^missile-mid-body-length 2.0)
                                         )
                                      0.0
                                      0.0)
                            (list (/ ^^missile-mid-body-length 2.0) 0.0 0.0)
                            )
             reference-coordinate-system ^^nose-coordinate-system
             )

            (mid-body :class 'open-cylindrical-body-class
              diameter       (* ^^missile-mid-body-radius 2.0)
              height         ^^missile-mid-body-length
              orientation    (list
                                 (rotate 90 '(0 1 0))
                                 )
              reference-coordinate-system ^^mid-body-coordinate-system
              )

            (aft-body-coordinate-system :class 'coordinate-system-class
              origin     (list
                             (+ (/ ^^missile-mid-body-length 2.0)
                                (/ ^^missile-aft-body-length 2.0)
                                 )
                             0.0
                             0.0)
              reference-coordinate-system ^^mid-body-coordinate-system
             )

            (aft-body :class 'open-truncated-cone-body-class
              start-diameter     (* ^^missile-aft-body-start-radius 2.0)
              end-diameter       (* ^^missile-aft-body-end-radius 2.0)
              height             ^^missile-aft-body-length
              orientation        (list
                                     (rotate 90 '(0 1 0))
                                     )
              reference-coordinate-system ^^aft-body-coordinate-system
              )
            )
)
```

Code Explanation

The *missile-geometry-class* now gives the user the ability to choose a nose type. The nose's class depends on the *missile-nose-type* property via a *case* statement. Notice the *nose-coordinate-system origin* property also depends on the *missile-nose-type* with an *if* condition in order to keep the nose tip aligned with the *missile-coordinate-system*. The *mid-body* and *aft-body* subobjects inherit from a separate class to allow similar functionality for future development as used in the nose-type options. The developer could provide other classes to instantiate with a dependency on a *missile-mid-body-type* property or *a missile-aft-body-type* property. This shows the extensibility of AML and some typical programming practices for enhancing future code.

The *nose* subobject uses the *default* functionality in the *missile-nose-radius* and *missile-nose-length* properties. The code could also be written as follows:

```
(nose :class (case !missile-nose-type
                (sphere   'spherical-nose-class)
                (cone     'open-conical-nose-class)
                (t        'spherical-nose-class)
                )
  missile-nose-radius  ^^missile-nose-radius
  missile-nose-length  ^^missile-nose-length
  reference-coordinate-system ^^nose-coordinate-system
 )
```

However, this would always add the *missile-nose-length* property to the *nose* object when it is an instance of *spherical-nose-class* class when it is not needed. This also eliminates the need to re-specify the relationships with the data model properties "above".

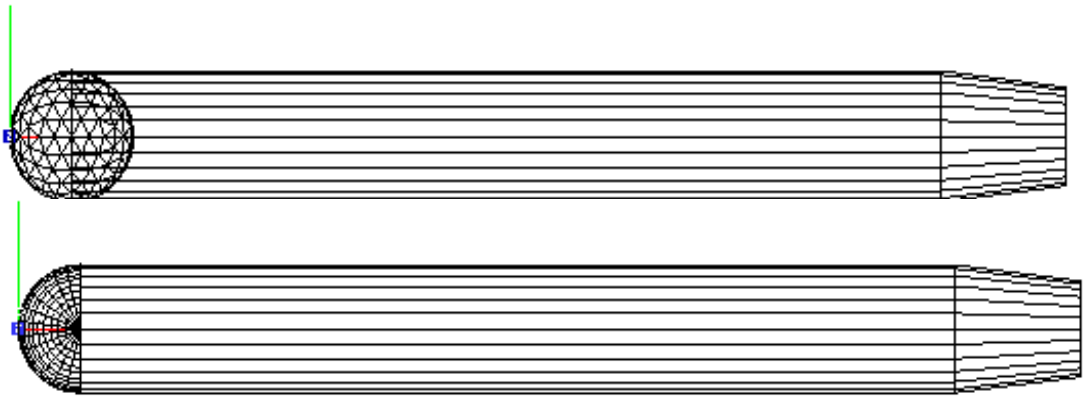
The *missile-nose-coordinate-system origin* could use a *case* statement depending on the *missile-nose-type* instead of using an *if* statement. The *case* statement may promote more robust code if other nose types are added in the future.

Notice how the model behaves when changing the *missile-nose-type* property. Simply pressing the "*regen*" button will not draw the nose after the nose type has been changed. This occurs because the nose instance has been "smashed", also known as being made "unbound". At this point, the complete nose instance is invalid, versus some particular geometry related properties being invalid. The "*regen*" command only regenerates objects in the graphics window where the properties that are needed for drawing have been smashed. To remedy this, the user should "*draw"* the nose object again after changing the nose type.

✓ Properties should not refer to objects or properties which are located outside of their class' scope. The class should be able to be instantiated in a "stand-alone" state allowing it to be placed in the model heirarchy at any location.

2.7 Geometric Booleans Enhance the Missile Geometry Class

The next step in creating the missile geometry is to augment the missile's design components so they are all thin shelled objects. In this case study the objects are modeled in a simplified manufacturing representation. The current geometry model is not accurate because the spherical nose is positioned inside of the mid-body. The next exercise introduces geometric booleans to eliminate the unneeded geometry. Figure 12 shows the comparison of the nose geometry before and after a difference operation with the mid-body.



Geometry Comparison of the nose boolean

Figure 12

New AML Constructs

The following AML concepts and constructs are used in this section:

- expanded property specification
- construction objects
- *cylinder-object*
- *difference-object*

Expanded Property Specification

✓ Properties are also objects. They can have properties themselves and also inherit from any predefined AML class. The specification is exactly like the subobject specification. For related information, please see the PROPERTY OBJECTS section of the AML Reference Manual.

All properties specified using the abbreviated syntax are created as objects of class *property-object*. This class has two properties: *formula* and *value*. In the abbreviated syntax, you are actually specifying the formula property of a property-object.

Format:

```
(name
   :class class-name
    prop-1       formula-1
     …
    prop-n       formula-n
  )
```

Arguments:

name	The name of the subobject.
class-name	A predefined (by the user, or inherently defined) AML class, for example: *'box-object*.
prop-x	The name of a property defined on the *class-name*.
formula-x	The formula which is replacing the *define-class* formula for property *prop-x*.

Example:

```
;;defined using brief form of property specification
(define-class EXAMPLE-MODEL-CLASS
    :inherit-from (object)
    :properties (
        category 'example
        code (if (equal ^category 'example)
                 45242
                 45789
                )
        )
    )

;;redefined using expanded form of property specification
(define-class EXAMPLE-MODEL-CLASS
    :inherit-from (object)
    :properties (
        (category :class 'property-object
            formula 'example
            )
        (code :class 'property-object
            formula (if (equal ^category 'example)
                        45242
                        45789
                    )
            value 45111
            )
        )
    )
```

✓ Please see the *difference-object* explanation below for an additional example of this functionality used with construction objects.

CYLINDER-OBJECT [Class]

A *cylinder-object* can be either a solid cylinder or a hollow cylindrical shell. It is hollow if the *solid?* property evaluates to *nil* and solid otherwise.

Inherit-From:

open-cylinder-object, solid-object

Properties:

diameter	The diameter of the cylinder. Default 1.0.
height	The height is defined parallel to the z-axis. Default 2.0.

CONSTRUCTION OBJECTS

This is a term commonly employed to refer to objects/subobjects that are needed to create the resultant geometry of booleans, for example.

DIFFERENCE-OBJECT [Class]

The *difference-object* takes a list of objects in the *object-list*. Starting with the first object in the *object-list*, it removes the parts of its geometry which intersect with the subsequent objects in the list.

Properties:

object-list	The list of objects which are to be differenced. Note that the order of the objects determines the final geometry.

Example:

```
(define-class TUBE-CLASS
:inherit-from (difference-object)
:properties (
            inner-diameter 0.5
            outer-diameter 1.0
            height 1.0
            render 'shaded

            object-list (list ^stock ^hole)

        (stock :class 'cylinder-object
            height ^^height
            diameter ^^outer-diameter
            solid? t
            )
        (hole :class 'cylinder-object
            height ^^height
            diameter ^^inner-diameter
```

```
                solid? t
                )
            )
        )
```

See Also:

Union-object
Trim-object
Sewn-object
Assembly-object
Group-object
Intersection-object
Divide-object
Sub-geom-object

Exercise 5

After a close review of the class definitions and AML constructs described in the previous section, augment the *spherical-nose-class* to inherits from a *difference-object*. The augmented *spherical-nose-class* has two construction objects in its properties, one named *complete-sphere* is of class *sphere-object* and the other named *missile-body-object* is of class *cylinder-object*. The *missile-body-object* is a solid cylinder so that the portion of the *complete-sphere* which lies inside the *missile-body-object* is removed. Add a property to the *missile-geometry-class* called *display-coord-systems?*. This property is used to control the ability of the coordinate systems in the subobjects to be drawn. Add a corresponding *the* reference in each of the coordinate system subobjects' *display?* property to depend on this property. Figure 13 shows a portion of the instance diagram for this exercise.

Instance Diagram for Exercise 5

```
spherical-nose-class [difference-object]
   missile-nose-radius
   object-list
   complete-sphere [sphere-object]
        diameter
   missile-body-object [cylinder-object]
        diameter
        height
        orientation
```

Figure 13

Exercise 5 Solution

```
(in-package :aml)

(define-class spherical-nose-class
  :inherit-from (difference-object)
  :properties (
               ;;user defined properties
               missile-nose-radius (default 1.0)

               ;;internal properties
               object-list (list ^complete-sphere ^missile-body-object)
               (complete-sphere :class 'sphere-object
                diameter    (* ^^missile-nose-radius 2.0)
                solid?      nil
                )
               (missile-body-object :class 'cylinder-object
                height      (* ^^missile-nose-radius 2.0)
                            ;;ensures the cylinder is large enough
                diameter    (* ^^missile-nose-radius 2.0)
                orientation (list
                             (rotate 90 '(0 1 0))
                             (translate (list (/ ^height 2.0) 0 0))
                             )
                )
               )
  )

(define-class open-conical-nose-class
  :inherit-from (open-cone-object)
  :properties (
               ;;user defined properties
               missile-nose-length (default 2.0)
               missile-nose-radius (default 1.0)

               ;;internal properties
               height              ^missile-nose-length
               diameter            (* ^missile-nose-radius 2.0)
               orientation         (list
                                    (rotate -90 '(0 1 0))
                                    )
               )
  )

(define-class open-cylindrical-body-class
  :inherit-from (open-cylinder-object)
  :properties (

               )
  )

(define-class open-truncated-cone-body-class
  :inherit-from (open-truncated-cone-object)
  :properties (

               )
  )

(define-class missile-geometry-class
  :inherit-from (object)
  :properties (
        ;;user defined properties
        missile-general-body-radius      (default 1.0)

        missile-nose-length   (default 2.0)
```

```
      missile-nose-radius   (default ^missile-general-body-radius)
      missile-nose-type     (default 'sphere)
                            ;; options are 'sphere or 'cone

      missile-mid-body-length (default 14.0)
      missile-mid-body-radius (default ^missile-general-body-radius)

      missile-aft-body-length       (default 2.0)
      missile-aft-body-start-radius (default ^missile-general-body-radius)
      missile-aft-body-end-radius   (default 0.75)

      display-coord-systems?        (default nil)

            )
:subobjects (
            (missile-coordinate-system :class 'coordinate-system-class
             display? ^^display-coord-systems?
             origin (list 0.0 0.0 0.0)
             )

            (nose-coordinate-system :class 'coordinate-system-class
              display? ^^display-coord-systems?
              origin  (if (equal ^^missile-nose-type 'cone)
                          (list (* 0.5 ^^missile-nose-length) 0.0 0.0)
                          (list ^^missile-nose-radius 0.0 0.0)
                          )
              reference-coordinate-system ^^missile-coordinate-system
             )

            (nose :class (case !missile-nose-type
                           (sphere   'spherical-nose-class)
                           (cone     'open-conical-nose-class)
                           (t        'spherical-nose-class)
                           )
              reference-coordinate-system ^^nose-coordinate-system
             )

            (mid-body-coordinate-system :class 'coordinate-system-class
             display? ^^display-coord-systems?
             origin  (if (equal ^^missile-nose-type 'cone)
                         (list (+ (/ ^^missile-nose-length 2.0)
                                  (/ ^^missile-mid-body-length 2.0)
                                  )
                               0.0
                               0.0)
                         (list (/ ^^missile-mid-body-length 2.0) 0.0 0.0)
                         )
             reference-coordinate-system ^^nose-coordinate-system
             )

            (mid-body :class 'open-cylindrical-body-class
              diameter       (* ^^missile-mid-body-radius 2.0)
              height         ^^missile-mid-body-length
              orientation    (list
                                 (rotate 90 '(0 1 0))
                                 )
              reference-coordinate-system ^^mid-body-coordinate-system
              )

            (aft-body-coordinate-system :class 'coordinate-system-class
              display? ^^display-coord-systems?
              origin     (list
                             (+ (/ ^^missile-mid-body-length 2.0)
                                (/ ^^missile-aft-body-length 2.0)
                                 )
                             0.0
                             0.0)
```

```
                    reference-coordinate-system ^^mid-body-coordinate-system
                   )

                  (aft-body :class 'open-truncated-cone-body-class
                    start-diameter     (* ^^missile-aft-body-start-radius 2.0)
                    end-diameter       (* ^^missile-aft-body-end-radius 2.0)
                    height             ^^missile-aft-body-length
                    orientation        (list
                                           (rotate 90 '(0 1 0))
                                           )
                    reference-coordinate-system ^^aft-body-coordinate-system
                    )
                  )
      )
```

Code Explanation

Notice how the *spherical-nose-class* does not have any subobjects. The complete-sphere and missile-body-object are considered "object properties". They are more of a means to get to an end product, rather than the end product. They are construction objects used to create the final *difference-object*. These objects can be considered as a mechanism that is needed to achieve the final geometry just as a height, width, and depth are needed to draw a box.

The subobjects of a class should typically not be used for utility operations. Subobjects should be used to represent parts of a whole assembly. For example, think of modeling a bearing with its internal components. The main class "bearing" would have subobjects "inner-race", "outer-race", "rollers", and "cage". Each of these "sub parts" creates the whole bearing object, therefore they should be subobjects.

When drawn (and expanded) an instance of a *spherical-nose-class* doesn't have subobjects. The object only has properties. Some of the properties are instances of *property*-object and others are geometric objects. Note that object properties are never drawn when their parent is drawn. They may, however, be drawn by specifically calling a *draw* method on them.

At this point in our modeling, we may not want to see the coordinate systems of the missile components. Therefore, the *display-coord-systems?* property is added at the top-level of the *missile-geometry-class* to control their *display?* property from one location. Some may argue that the coordinate systems are construction objects that are used to build and orient the components. In some cases this is true, but usually, a coordinate system is useful to have displayed in a part tree and to have the ability to draw it easily. Notice the practice of having a question mark at the end of any property that evaluates to *t* or *nil*.

2.8 Creation of a Fin Profile, Fin Extrusion, and Fin Array

The next step in creating the missile geometry is to develop class definitions for aerodynamic control fins. In this case study, the fins are very simple and used for demonstration purposes. The fins are produced from a hexagonal profile that is extruded and then arrayed circularly. Figure 14 shows the fin profile and its dimensions, Figure 15 shows the extruded profile, and Figure 16 shows the extruded fin in a circular array.

Fin Profile

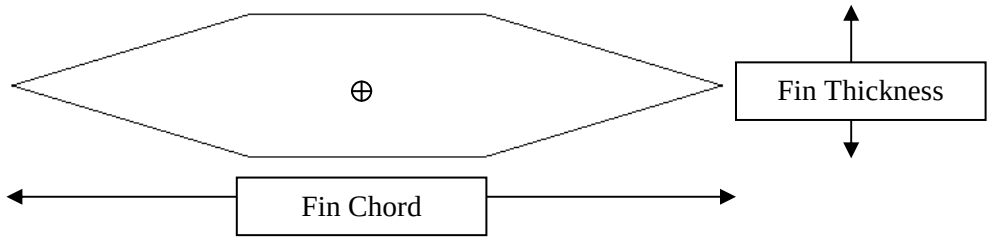


Figure 14

Fin Extrusion

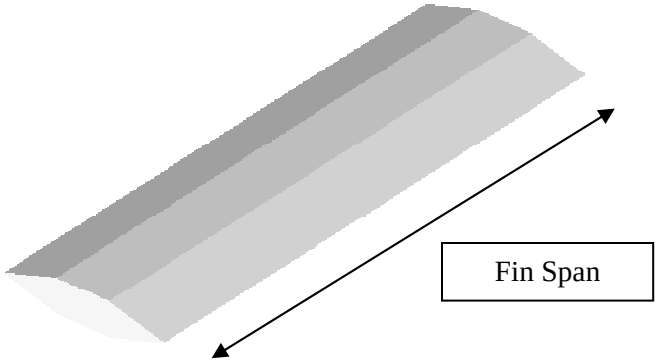


Figure 15

Fin Circular Array

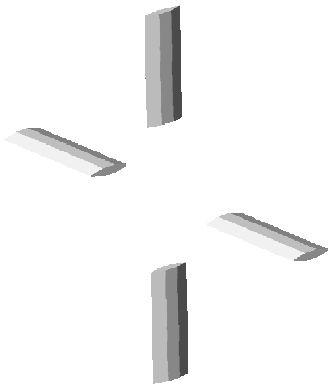


Figure 16

New AML Constructs

The following AML concepts and constructs are used in this section:

- *let**
- *polygon-object*
- *extrusion-object*
- *capped-surface-object*
- *circular-array-object*
- *name-generator*
- *generate-name*
- *add-object*

The *let** form is the most common mechanism for creating local variables. The *let** construct binds values to the variables in sequence which means that a variable may use a variable previously defined in the same *let** variable assignment.

Format:

(LET* (var-assignments) body)

Arguments:

var-assignments	The let clauses are a list of variable formula lists and/or variables.
body	Any number of expressions to be evaluated within the context of the assigned variables. The return value of the last expression will be returned from the *let** function.

Examples:

```
AML> (LET* ((a    3.0)
            (b    (* 2 a))
            )
       (+ 1.0 b)
       )
7.0
AML> (LET* ((a    3.0)
            (b    (* 2 a))
            )
       (print a)
       (/ b a)
       (print b)
       (+ 1.0 b)
       )
3.0
6.0
7.0
```

✓ Note that order matters in the variable assignment clauses. As seen in the previous example, *a* must be declared before *b*.

The following example shows an implementation of the quadratic formula. The *expt* function raises the first argument to the power given in the second argument. The *sqrt* function takes the square root of the first argument.

```
AML> (LET* ((a                1)
            (b                3)
            (c                2)
            (radical          (- (expt b 2) (* 4 a c)))
            (denominator      (* 2 a))
            (numerator-plus   (+ (- b) (sqrt radical)))
            (numerator-minus  (- (- b) (sqrt radical)))
            )
```

```
            (list
                (/ numerator-plus denominator)
                (/ numerator-minus denominator))
            )
(-1.0 -2.0)
```

POLYGON-OBJECT [Class]

The *polygon-object* is a polygon composed of line segments. If the vertices do not specify a closed polygon, the system will automatically connect the start and end points.

Properties:

vertices	The list of points defining the segments of the polygon-object. Default *nil*. These vertices must be planar if *dimension* is 2.
dimension	The dimension determines if the object is a 1d entity (an outline) or a 2d entity (a surface). The default value is 1 for 1d.

Example:

```
(define-class EXAMPLE-POLYGON-CLASS
    :inherit-from (polygon-object)
    :properties (
        vertices    '(
                        (0.0 0.0 0.0)
                        (1.0 0.0 0.0)
                        (1.0 1.0 0.0)
                        (0.0 1.0 0.0)
                        )
        dimension   2
            )
    )

;;demonstrates the use of let* in a property
(define-class RECTANGLE-CLASS
    :inherit-from (polygon-object)
    :properties (
        x-dim       (default 1.0)
        y-dim       (default 1.0)

        vertices    (let* (
                           (half-x (* 0.5 ^x-dim))
                           (half-y (* 0.5 ^y-dim))
                           )
                       (list
                           (list half-x half-y 0.0)
                           (list (- half-x) half-y 0.0)
                           (list (- half-x) (- half-y) 0.0)
                           (list half-x (- half-y) 0.0)
                           )
                       )
        dimension   2
        )
    )
```

EXTRUSION-OBJECT [Class]

The *extrusion-object* sweeps an object along a vector. The resulting geometry may be a solid or surface and capped or not capped. When the geometry is solid it must be capped.

Properties:

swept-object	The object that is to be extruded.
vector	The list of x, y, and z vector components of the direction to extrude.
distance	The length of extrusion.

Example:

```
(define-class RECTANGLE-CLASS
    :inherit-from (polygon-object)
    :properties (
        x-dim       (default 1.0)
        y-dim       (default 1.0)

        vertices    (let* (
                            (half-x (* 0.5 ^x-dim))
                            (half-y (* 0.5 ^y-dim))
                            )
                        (list
                            (list half-x half-y 0.0)
                            (list (- half-x) half-y 0.0)
                            (list (- half-x) (- half-y) 0.0)
                            (list half-x (- half-y) 0.0)
                            )
                        )
        dimension   2
        )
    )

(define-class EXAMPLE-BAR-EXTRUSION-CLASS
    :inherit-from (extrusion-object)
    :properties (
        x-dim       (default 1.0)
        y-dim       (default 1.0)
        bar-length  (default 10)

        swept-object        ^profile
        vector              '(0.0 0.0 1.0)
        distance            ^bar-length

        (profile :class 'rectangle-class)
         )
    )
```

CAPPED-SURFACE-OBJECT [Class]

A *capped-surface-object* takes an open surface (such as an *open-cylinder-object*, or a *skin-surface-from-curves-object*, ...) as a source-object, and creates either a capped surface (solid? = nil) or a solid bounded by the capped surface (solid? = t).

Properties:

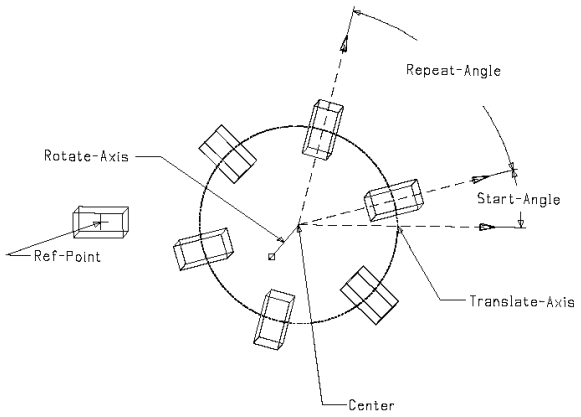
source-object	An instance of the open surface to cap.
solid?	Default *t* to create a capped solid. When *nil*, the surface is only capped and no solid geometry is created.

CIRCULAR-ARRAY-OBJECT [Class]

The array is created by making copies of the geometry of the *source-object* and placing them in a circle around the *center* at the given *diameter*. The circle is normal to the *rotate-axis*. The first copy is translated along the *translate-axis* and then rotated about the *rotate-axis* counter-clockwise by the *start-angle*. Subsequent copies are each placed at a further counter-clockwise position incremented by *repeat-angle*. The *circular-array-object* creates a single geometric *assembly-object*. The array is similar to the *circular-clonified-object* object except that there is only one object with a representation that is the complete array geometry. If individual geometry is required, use the *circular-clonified-object*.

Properties:

source-object	The object to be copied.
diameter	The size of the circle of which the copies will be placed about. Default 1.0.
start-angle	The angle at which to place the first array element (with reference to the *translate-axis*). Default 0.0.
repeat-angle	The angular spacing between two consecutive array elements. Default *(/ 360 !quantity)*
translate-axis	The reference vector for positioning the first copy. Default is the x-axis.
rotate-axis	Defines the axis perpendicular to the plane of the circular array. Default is the y-axis.
rotate-clones?	Default *t*, it decides whether or not the copies will be individually rotated as they are laid out in the circular pattern.
center	The center of the circular pattern, it defaults to the center of the *source-object*.
quantity	The number of copies to create. Default 1.
assembly?	*t* for creating an assembly and *nil* for creating a union. The default value is *t*.
ref-point	The local point on the *source-object* (and each copy) that lies on the described circle. It defaults to the center of the *source-object*.



See Also:

```
circular-clonified-object
linear-clonified-object
linear-array-object
series-object
```

✓ The following class and methods are introduced to promote the use of dynamic object addition. As previously stated, the *create-model* function can be used to instantiate a class. However, a typical application will have the concept of a model as the mechanism to encapsulate an entire design. Thus an application may have one, two, or three models and could have hundreds or thousands of objects within those models. In addition, *create-model* performs other functions such as selecting the current model and setting the *the* tracing to the currently created model. Therefore the *name-generator* class and the *add-object* method are shown and their use is encouraged now. Even though the AML construct *define-method* is not introduced until the "Optional Topics" portion of this manual, the user will use methods throughout the training course such as *draw*, *shade*, and *expand*.

NAME-GENERATOR [Class]

This class provides the ability to generate unique names by appending numbers to the end of the names. This can be used when the user adds several objects to a model without specifically naming each object manually (or writing an algorithm to do this). Especially for debugging, a user can create a model of type *name-generator* and dynamically add objects to this model without having to specifically name all of the objects. The *name-generator* class has methods to automatically perform and manage the naming such as *generate-name*.

Properties:

previous-name-list

This property is used to store the names and current numbers of names that have been generated. Default *nil*.

auto-naming? A value of *t* will cause the *generate-name* method to automatically generate the next name for the prefix specified. A value of *nil* will cause the *generate-name* method to prompt the user to enter a name. Default *t*.

GENERATE-NAME [Method]

The *generate-*name method is used to append an incremental suffix to the name supplied.

Format:

(GENERATE-NAME instance name)

Arguments:

Instance	An instance of a *name-generator* class.
name	The base name to which the numbered suffix is appended.

Example:

```
AML> (create-model 'name-generator)
#<NAME-GENERATOR @ #x117792a>
AML> (generate-name (the) 'box)
BOX-0001
AML> (generate-name (the) 'box)
BOX-0002
AML> (generate-name (the) 'box)
BOX-0003
AML> (generate-name (the) 'cylinder)
CYLINDER-0001
AML> (generate-name (the) 'cylinder)
CYLINDER-0002
AML> (generate-name (the) 'cylinder)
CYLINDER-0003
AML> (generate-name (the) 'box)
BOX-0004
AML> (add-object (the) (generate-name (the) 'box) 'box-object)
#<BOX-OBJECT @ #x1c07552>
AML> (object-name *)
BOX-0005
```

Note: When at the AML command prompt, the * symbol returns the return value from the function/method previously called. This can be used as an argument to AML functions/methods.

ADD-OBJECT [Method]

Add-object instantiates a class and adds the object to the object instance specified in the *parent* argument with the name given in the *name* argument.

Format:

(ADD-OBJECT parent name class)

Arguments:

parent	The object instance that the new object will be added to.
name	The name of the object instance to be added.
class	The class of the object instance to be added.

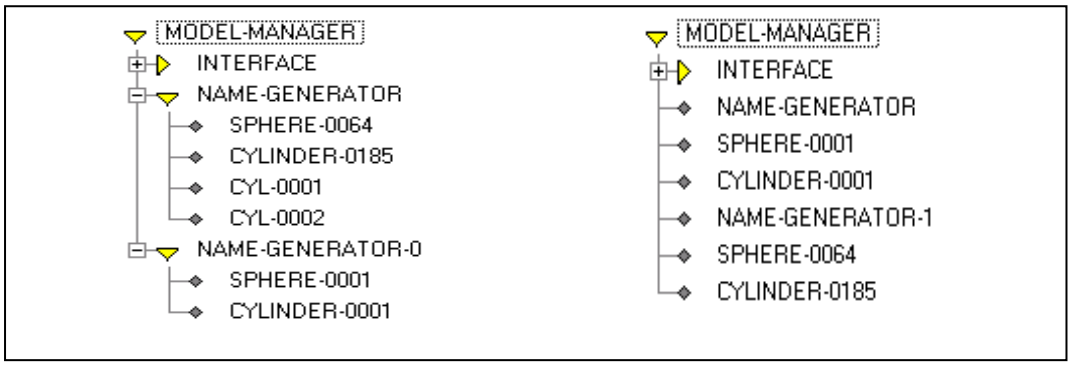
Examples:

```
AML> (create-model ‘name-generator)
#<NAME-GENERATOR @ #x117792a>
AML> (add-object (the) ‘sphere-0001 ‘sphere-object)
#<SPHERE-OBJECT @ #x522567a>
AML> (add-object (the) ‘cylinder-0001 ‘open-cylinder-object)
#<OPEN-CYLINDER-OBJECT @ #x392887a>
AML> (create-model ‘name-generator)
#<NAME-GENERATOR @ #x182579f>
AML> (add-object (the) ‘sphere-0064 ‘sphere-object)
#<SPHERE-OBJECT @ #x21f3f7a2>
AML> (add-object (the) ‘cylinder-0185 ‘open-cylinder-object)
#<OPEN-CYLINDER-OBJECT @ #x21f5d5b2>
AML> (add-object (the) (generate-name (the)‘cyl) ‘open-cylinder-object)
#<OPEN-CYLINDER-OBJECT @ #x21f5f1b1>
AML> (add-object (the) (generate-name (the)‘cyl) ‘open-cylinder-object)
#<OPEN-CYLINDER-OBJECT @ #x44b5d511>
```

Whereas performing the following commands would create a different tree structure.

```
AML> (create-model ‘name-generator)
#<NAME-GENERATOR @ #x117792a>
AML> (create-model ‘sphere-0001 :class ‘sphere-object)
#<SPHERE-OBJECT @ #x522567a>
AML> (create-model ‘cylinder-0001 :class ‘open-cylinder-object)
#<OPEN-CYLINDER-OBJECT @ #x392887a>
AML> (create-model ‘name-generator)
#<NAME-GENERATOR @ #x182519f>
AML> (create-model ‘sphere-0064 :class ‘sphere-object)
#<SPHERE-OBJECT @ #x21f3f3a2>
AML> (create-model ‘cylinder-0185 :class ‘open-cylinder-object)
#<OPEN-CYLINDER-OBJECT @ #x21f9d5b2>
```

Notice the differences in the tree structures below. The *create*-model commands will automatically rename the first instance of the *name-generator* as seen below:



✓ Notice how the two methodologies of instantiation create different results. The process of creating models should be considered as if the model represents one version of a complete system/application. The process of adding objects allows a user to dynamically build an instance hierarchy. Typically a user/developer would use *create-model* to create one version of the application and then utilize *add-object* to create the bulk of an application underneath one model. Also note that when using *create-model*, a user/developer would then have to use *select-model* to change between the various models whereas with *add-object*, all objects can be located within the same model.

✓ Notice how the *generate-name* method automatically generates a name based on a given prefix. The trainee is encouraged to refer to the AML Reference Manual for a further description of *generate-name*.

Exercise 6

After a close review of the class definitions and AML constructs described in the previous section, create the following class definitions:

- hexagonal-profile-class [*polygon-object*]
- fin-profile-class [*hexagonal-profile-class*]
- fin-class [*capped-surface-object*]
- fin-array-class [*circular-array-object*]

Use the *let** construct in the formula of the *vertices* in the *hexagonal-profile-class*. The dimensions for the *hexagonal-profile-class* assume that each panel's chord projection to the x-axis is 1/3 of the total chord. The *vertices* in the *hexagonal-profile-class* should depend on a generic *height* and *width* property with respective default values of 1.0 and 0.2. This class is the base class inherited into a more specific *fin-profile-class* which adds some "fin-specific" properties called *fin-chord* and *fin-thickness* that feed the generic *width* and *height* properties respectively.

The *fin-class* inherits from a *capped-surface-object* and contains two object properties called *fin-profile* and *fin-extrusion*. The *fin-extrusion* uses the *fin-profile* as a *swept-object*. The *fin-class* has a property called *fin-span* that specifies the length of the fin.

The *fin-array-class* inherits from *circular-array-class* and uses an instance of a *fin-class* as a source object. The *diameter* should be twice the size of a user-defined property called *fin-span-offset* which will later be set equal to the *missile-general-body-radius*. Also give the *fin-array-class* a property called *fin-quantity* to specify the number of fins in the array.

✓ Note that all of the *fin-xxxxx* properties are given (repeated) in each of the classes leading up to the *fin-class*. This is shown here to demonstrate the practice of creating classes that can stand alone and are modular.

Instead of performing a *create-model* operation to make a new instance of the *missile-geometry-class*, create a model of a *name-generator* and then continue to add new instances of the *missile-geometry-class* to the *name-generator* object using *add-object* as you continue in the training course. The instructor will demonstrate how to perform this through the user interface. The user interface will automatically name the objects as you add them by calling the *generate-name* method on the *name-generator*. Figure 17 shows the instance diagram.

Instance Diagram for Exercise 6

```
hexagonal-profile-class [polygon-object]
    width
    height
    vertices

fin-profile-class [hexagonal-profile-class]
    fin-cord
    fin-thickness
    width
    height
```

```
    fin-class [capped-surface-object]
        fin-cord
        fin-thickness
        fin-span
        solid?
        source-object
        fin-profile [fin-profile-class]
        fin-extrusion [extrusion-object]
            swept-object
            distance
            vector

fin-array-class [circular-array-object]
        fin-cord
        fin-thickness
        fin-span
        fin-span-offset
        fin-quantity
        source-object
        diameter
        quantity
        ref-point
        center
        rotate-axis
        fin-source [fin-class]
            orientation
```

Figure 17

Exercise 6 Solution

```
(in-package :aml)

(define-class hexagonal-profile-class
  :inherit-from (polygon-object)
  :properties (
               ;;user defined properties
               width     (default 1.0)
               height    (default 0.2)

               ;;internal properties
               vertices  (let* (
                                (half-w   (/ ^width 2.0))
                                (half-h   (/ ^height 2.0))
                                (sixth-w  (/ ^width 6.0))
                                )
                           (list
                            (list half-w 0.0 0.0)
                            (list sixth-w half-h 0.0)
                            (list (- sixth-w) half-h 0.0)
                            (list (- half-w) 0.0 0.0)
                            (list (- sixth-w) (- half-h) 0.0)
                            (list sixth-w (- half-h) 0.0)
                            ;;last point will be closed automatically
                            )
                           )
                )
  )

(define-class fin-profile-class
  :inherit-from (hexagonal-profile-class)
  :properties (
               ;;user defined properties
               fin-cord          (default 1.0)
               fin-thickness     (default 0.2)

               ;;internal properties
               width             ^fin-cord
               height            ^fin-thickness
               )
  )

(define-class fin-class
  :inherit-from (capped-surface-object)
  :properties (
               ;;user defined properties
               fin-cord          (default 1.0)
               fin-thickness     (default 0.2)
               fin-span          (default 3.0)

               ;;internal properties
               solid?            nil
               source-object     ^fin-extrusion

               (fin-profile :class 'fin-profile-class

                )

               (fin-extrusion :class 'extrusion-object
                swept-object      ^^fin-profile
                distance          ^^fin-span
                vector            '(0.0 0.0 1.0)
                )
               )
  )
```

```
(define-class fin-array-class
  :inherit-from (circular-array-object)
  :properties (
               ;;user defined properties
               fin-cord              (default 1.0)
               fin-thickness         (default 0.2)
               fin-span              (default 3.0)
               fin-span-offset       (default 2.0)
               fin-quantity          (default 4)

               ;;internal properties
               source-object      ^fin-source
               diameter           (* 2.0 ^fin-span-offset)
               quantity           ^fin-quantity
               ref-point          '(0.0 0.0 0.0)
               center             '(0.0 0.0 0.0)
               rotate-axis        '(0.0 0.0 1.0)

               (fin-source :class 'fin-class
                 orientation  (list
                               (rotate 90.0 '(0 1 0))
                               )
                )
               )
  )
```

Code Explanation

The developer creates a generic *hexagonal-profile-class* before creating the *fin-profile-class*. This promotes code re-use and modular design as the generic class can be instantiated by itself or inherited into other user-defined classes in the future. The *let** construct organizes the formula for the *vertices* to make the code more readable and efficient. If the code did not have the *let**, the computer would have to evaluate several exactly equal calculations repeatedly, thus making the code inefficient. On the other hand, a developer should not try to use too many variables in a *let** if they are not necessary because that will cause unnecessary memory allocation.

2.9 Missile Geometry with a Fin Array and Material Properties

Figure 18 shows the missile geometry with a fin array. The fin array is oriented with respect to the *missile-coordinate-system* in this case study. This section of the manual also introduces some additional properties to the missile components via a *material-properties-class*. This is a user-defined class that is extensible beyond the scope covered in this case study. The class simply gives each component a material name, material thickness, and material density. This class demonstrates a use of multiple inheritance.

Missile Geometry with a Fin Array

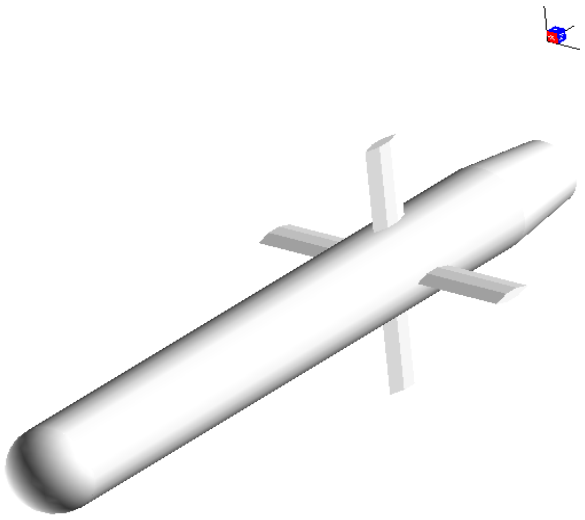


Figure 18

New AML Constructs

The following AML concepts and constructs are used in this section:

- multiple-inheritance

Multiple Inheritance

Inheritance is a mechanism for class reuse. Through inheritance, a class will have all of the same properties, subobjects, and methods as the classes that it inherits from (its superclasses). The :*inherit-from* keyword accepts a list of classes to be used as superclasses. If a property, subobject, or method is present in more than one of the superclasses, the order of precedence is from left to right. If an :*inherit-from* statement contains the list *(box-object cylinder-object), the box-object* geometry will be created. If the order of the list is reversed to *(cylinder-object box-*

object), the *cylinder-object* geometry will be created. **Note: It is not good code writing practice to inherit from two classes that have *graphic-object* as a superclass.**

Example:

```
(define-class material-class
  :inherit-from (object)
  :properties (
               density   1.0
               material 'wood
               color    'blue
               )
  )

(define-class multiple-inheritance-example-class
  :inherit-from (material-class box-object)
  :properties (
               depth     2.0
               width     3.0
               height    1.0

               density   2.649
               )
  )
```

AML uses the concept of "first come, first serve". Thus the class given first in the *:inherit-from* statement will take precedence over any classes given after. Eventhough they are not shown, the *multiple-inheritance-example-class* has properties of *color* (set to 'blue) and *material* (set to 'wood). This can be demonstrated using the *describe* function at the command prompt as follows.

```
> (create-model 'multiple-inheritance-example-class)
#<MULTIPLE-INHERITANCE-EXAMPLE-CLASS @ #x22f5bc2a>
> (describe (the))

DEFINE-CLASS PROPERTIES:
------------------------
 DEPTH                      | Unbound |
 COLOR                      | Unbound |
 LAYER                      | Unbound |
 GEOM                       | Unbound |
 RENDER                     | Unbound |
 REFERENCE-OBJECT           | Unbound |
 POSITION                   | Unbound |
 LINE-TYPE                  | Unbound |
 SOLID?                     | Unbound |
 DENSITY                    | Unbound |
 WIDTH                      | Unbound |
 DISPLAY?                   | Unbound |
 REFERENCE-COORDINATE-SYSTEM | Unbound |
 MATERIAL                   | Unbound |
 HEIGHT                     | Unbound |
 LINE-WIDTH                 | Unbound |
 ORIENTATION                | Unbound |

DEFINE-CLASS SUBOBJECTS:
------------------------
NIL
> (the color)
BLUE
```

Exercise 7

After a close review of the AML constructs described in the previous section, create the following class definitions:

- material-properties-class [*object*]
- missile-body-component-class [*material-properties-class*]

The *material-properties-class* will have three properties: *material-thickness (default 0.01), material-name (default 'aluminum), material-density (default 165.0).*

Add a property called *missile-general-body-thickness* to the *missile-geometry-class*. Augment all of the body component classes so they inherit from *missile-body-component-class*, and update their respective instances by adding a *material-thickness* property which references the *missile-general-body-thickness*. Add an instance of *fin-array-class* to the *missile-geometry-class*. Orient this with respect to the *missile-coordinate-system* using a property in the *missile-geometry-class* called *fin-distance-along-axis* (use *(default 12.0)*). Add the necessary properties of a *fin-array-class* to the *missile-geometry-class* in order to best use the *default* functionality.

The user may choose to create a separate coordinate system for the *fin-array-class* instance for orientation and positioning, but this example uses the *missile-coordinate-system* as a *reference-coordinate-system.*

Exercise 7 Solution

```
(in-package :aml)

(define-class material-properties-class
  :inherit-from (object)
  :properties (
               ;;English units assumed
               material-thickness     (default 0.01)        ;ft
               material-name          (default 'aluminum)   ;symbol
               material-density       (default 165.0)       ;lb/ft^3
               )
  )

(define-class missile-body-component-class
  :inherit-from (material-properties-class)
  :properties (

               )
  )

(define-class spherical-nose-class
  :inherit-from (difference-object missile-body-component-class)
  :properties (
               ;;user defined properties
               missile-nose-radius (default 1.0)

               ;;internal properties
               object-list (list ^complete-sphere ^missile-body-object)
               (complete-sphere :class 'sphere-object
                diameter    (* ^^missile-nose-radius 2.0)
                solid?      nil
                )
               (missile-body-object :class 'cylinder-object
                ;;* 2.0 ensures the cylinder height is large enough
                height      (* ^^missile-nose-radius 2.0)
                diameter    (* ^^missile-nose-radius 2.0)
                orientation (list
                             (rotate 90 '(0 1 0))
                             (translate (list (/ ^height 2.0) 0 0))
                             )
                )
               )
  )

(define-class open-conical-nose-class
  :inherit-from (open-cone-object missile-body-component-class)
  :properties (
               ;;user defined properties
               missile-nose-length (default 2.0)
               missile-nose-radius (default 1.0)

               ;;internal properties
               height              ^missile-nose-length
               diameter            (* ^missile-nose-radius 2.0)
               orientation         (list
                                    (rotate -90 '(0 1 0))
                                    )
               )
  )

(define-class open-cylindrical-body-class
  :inherit-from (open-cylinder-object missile-body-component-class)
  :properties (
```

```
              )
  )

(define-class open-truncated-cone-body-class
  :inherit-from (open-truncated-cone-object missile-body-component-class)
  :properties (

              )
  )

(define-class hexagonal-profile-class
  :inherit-from (polygon-object)
  :properties (
              ;;user defined properties
              width     (default 1.0)
              height    (default 0.2)

              ;;internal properties
              vertices  (let* (
                              (half-w   (/ ^width 2.0))
                              (half-h   (/ ^height 2.0))
                              (sixth-w  (/ ^width 6.0))
                              )
                         (list
                          (list half-w 0.0 0.0)
                          (list sixth-w half-h 0.0)
                          (list (- sixth-w) half-h 0.0)
                          (list (- half-w) 0.0 0.0)
                          (list (- sixth-w) (- half-h) 0.0)
                          (list sixth-w (- half-h) 0.0)
                          ;;last point will be closed automatically
                          )
                         )
               )
  )

(define-class fin-profile-class
  :inherit-from (hexagonal-profile-class)
  :properties (
              ;;user defined properties
              fin-cord          (default 1.0)
              fin-thickness     (default 0.2)

              ;;internal properties
              width             ^fin-cord
              height            ^fin-thickness
              )
  )

(define-class fin-class
  :inherit-from (capped-surface-object)
  :properties (
              ;;user defined properties
              fin-cord          (default 1.0)
              fin-thickness     (default 0.2)
              fin-span          (default 3.0)

              ;;internal properties
              solid?            nil
              source-object     ^fin-extrusion

              (fin-profile :class 'fin-profile-class

               )
```

```
              (fin-extrusion :class 'extrusion-object
               swept-object      ^^fin-profile
               distance          ^^fin-span
               vector            '(0.0 0.0 1.0)
               )
              )
  )

(define-class fin-array-class
  :inherit-from (circular-array-object)
  :properties (
              ;;user defined properties
              fin-cord              (default 1.0)
              fin-thickness         (default 0.2)
              fin-span              (default 3.0)
              fin-span-offset       (default 2.0)
              fin-quantity          (default 4)

              ;;internal properties
              source-object      ^fin-source
              diameter           (* 2.0 ^fin-span-offset)
              quantity           ^fin-quantity
              ref-point          '(0.0 0.0 0.0)
              center             '(0.0 0.0 0.0)
              rotate-axis        '(0.0 0.0 1.0)
              repeat-angle       (/ 360.0 ^fin-quantity)

              (fin-source :class 'fin-class
                orientation  (list
                              (rotate 90.0 '(0 1 0))
                              )
               )
              )
  )

(define-class missile-geometry-class
  :inherit-from (object)
  :properties (
        ;;user defined properties
        missile-general-body-radius      (default 1.0)
        missile-general-body-thickness   (default 0.01)

        missile-nose-length   (default 2.0)
        missile-nose-radius   (default ^missile-general-body-radius)
        missile-nose-type     (default 'sphere)
                              ;; options are 'sphere or 'cone

        missile-mid-body-length (default 14.0)
        missile-mid-body-radius (default ^missile-general-body-radius)

        missile-aft-body-length       (default 2.0)
        missile-aft-body-start-radius (default ^missile-general-body-radius)
        missile-aft-body-end-radius   (default 0.75)

        fin-distance-along-axis       (default 12.0)
        fin-cord                      (default 1.0)
        fin-thickness                 (default 0.2)
        fin-span                      (default 2.0)
        fin-span-offset               (default ^missile-general-body-radius)
        fin-quantity                  (default 4)

        display-coord-systems?        (default nil)
```

```
                )
:subobjects (
               (missile-coordinate-system :class 'coordinate-system-class
                display? ^^display-coord-systems?
                origin (list 0.0 0.0 0.0)
                )

               (nose-coordinate-system :class 'coordinate-system-class
                 display? ^^display-coord-systems?
                 origin     (if (equal ^^missile-nose-type 'cone)
                                (list (* 0.5 ^^missile-nose-length) 0.0 0.0)
                                (list ^^missile-nose-radius 0.0 0.0)
                                )
                 reference-coordinate-system ^^missile-coordinate-system
                )

               (nose :class (case !missile-nose-type
                              (sphere   'spherical-nose-class)
                              (cone     'open-conical-nose-class)
                              (t        'spherical-nose-class)
                              )
                 material-thickness  ^^missile-general-body-thickness
                 reference-coordinate-system ^^nose-coordinate-system
                )

               (mid-body-coordinate-system :class 'coordinate-system-class
                display? ^^display-coord-systems?
                origin  (if (equal ^^missile-nose-type 'cone)
                            (list (+ (/ ^^missile-nose-length 2.0)
                                     (/ ^^missile-mid-body-length 2.0)
                                     )
                                  0.0
                                  0.0)
                            (list (/ ^^missile-mid-body-length 2.0) 0.0 0.0)
                            )
                reference-coordinate-system ^^nose-coordinate-system
                )

               (mid-body :class 'open-cylindrical-body-class
                 diameter       (* ^^missile-mid-body-radius 2.0)
                 height         ^^missile-mid-body-length
                 material-thickness  ^^missile-general-body-thickness
                 orientation    (list
                                    (rotate 90 '(0 1 0))
                                    )
                 reference-coordinate-system ^^mid-body-coordinate-system
                 )

               (aft-body-coordinate-system :class 'coordinate-system-class
                 display? ^^display-coord-systems?
                 origin     (list
                                (+ (/ ^^missile-mid-body-length 2.0)
                                   (/ ^^missile-aft-body-length 2.0)
                                    )
                                0.0
                                0.0)
                 reference-coordinate-system ^^mid-body-coordinate-system
                )

               (aft-body :class 'open-truncated-cone-body-class
                 start-diameter     (* ^^missile-aft-body-start-radius 2.0)
                 end-diameter       (* ^^missile-aft-body-end-radius 2.0)
                 height             ^^missile-aft-body-length
                 material-thickness ^^missile-general-body-thickness
                 orientation        (list
                                        (rotate 90 '(0 1 0))
                                        )
```

```
                    reference-coordinate-system ^^aft-body-coordinate-system
                    )
                  (fins :class 'fin-array-class
                    orientation  (list
                                  (rotate 90 '(0 1 0))
                                  (translate (list ^^fin-distance-along-axis
                                                   0.0
                                                   0.0)
                                      )
                                  )
                    reference-coordinate-system ^^missile-coordinate-system
                   )

                  )
)
```

Code Explanation

The developer creates a generic *material-properties-class* before creating the *missile-body-component-class*. This promotes code re-use and modular design as the generic class can be instantiated by itself or inherited into other user-defined classes in the future. The inheritance capability will provide advantages in the next exercises because we now have a common class from which all of the body components inherit. This also can be helpful when using class "filters" during object selections from graphic or tree.

2.10 Final Missile Geometry and Mass Properties

This section of the manual introduces some additional properties to the missile components via a *mass-properties-class*. This is a user-defined class that is extensible beyond the scope covered in this case study. The class gives each component a mass and a surface area property. When collected in a property at the top-level of the *missile-geometry-class*, this demonstrates the use of *loop*, and some geometric queries.

New AML Constructs

The following AML concepts and constructs are used in this section:

- *children*
- *volume-of-object*
- *when*
- *loop*
- *the* referencing (expanded explanation)

CHILDREN **[Method]**

Returns the immediate subobjects of an instance. Note that this method returns a list of objects, not the names of the objects

Format:

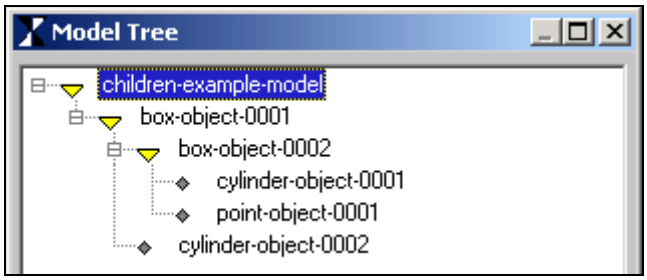
(CHILDREN instance [:class t])

Arguments:

instance	The object instance whose subobjects you wish to find.
:class	If a class is specified, only children inheriting from that class will be returned. Default is *t* for all classes. Note, a list of classes may also be specified here.

Example:

Assume that you have the following tree structure where each object inherits from the class of its name:




```
AML> (create-model 'children-example-model)
<CHILDREN-EXAMPLE-MODEL #x1B88768>
AML> (children (the))
(#<BOX-OBJECT #x1A68338>)
AML> (children (the box-object-1))
(#<BOX-OBJECT #x1A8FDB8> #<CYLINDER-OBJECT #x1AC32D8>)
AML> (children (the box-object-1 box-object-2) :class 'point-object)
(#<POINT-OBJECT #x1AB2048>)
```

Notice that the *children* method returns a list of object instances (handles to the objects).

See Also:

`children` ←(has more keywords)
`subobjects` ← method returns object names instead of object
`select-object` ← function to query/return all objects "under" an object

VOLUME-OF-OBJECT graphic-object [Method]

The value returned by this method depends on the dimension of the object. If the objects's dimension is 1, then the length is returned. If the objects's dimension is 2, then the surface area is returned. If the objects's dimension is 3, then the volume is returned.

Format:

(VOLUME-OF-OBJECT instance)

Arguments:

instance An instance of a graphic object.

Example:

```
(define-class EXAMPLE-BOX-CLASS
  :inherit-from (box-object)
  :properties (
               height  3.5
               width   4.1
               depth   (* 1.8 ^height)

               volume  (volume-of-object !superior)
                )
  )
```

```
AML> (create-model 'EXAMPLE-BOX-CLASS)
#<EXAMPLE-BOX-CLASS @ #x21a2975a>
AML> (the height)
3.5
AML> (the volume)
90.40500258922566
AML> (create-model 'BOX-OBJECT)
#<BOX-OBJECT @ #x28a4575b>
AML> (volume-of-object (the))
1.0

(define-class EXAMPLE-LINE-CLASS
  :inherit-from (line-object)
  :properties (
               point1  '(0 0 0)
               point2  '(2.4 6.7 8.4)

               length  (volume-of-object !superior)
                )
  )
AML> (create-model 'EXAMPLE-LINE-CLASS)
#<EXAMPLE-LINE-CLASS @ #x21a8dc7a>
AML> (the length)
11.009540930176211
AML> (create-model 'cylinder-object)
#<CYLINDER-OBJECT @ #x58b1fb2b>
AML> (volume-of-object (the))
1.570796326794895
```

See Also:

normal-to-face
center-of-object
dimension
several functions/classes to calculate tangency

---

when **[Function]**

The when statement is a conditional that evaluates a test and if the result is not nil then all of the expressions of the body are evaluated.

Format:

(WHEN test body)

Arguments:

test	Any expression that will be used to determine whether to evaluate the body.
body	Any number of expressions to be evaluated when test is true. The return value of the last expression is returned from the *when* function.

Examples:

```
AML> (when (= 2 3)
    (+ 2 3)
    (print (/ 6 3))
```

```
    (* 3 4)
    )
NIL
AML> (when (< 2 3)
    (+ 2 3)
    (print (/ 6 3))
    (* 3 4)
    )
2
12
AML> (when 'false
    (+ 2 3)
    (print (/ 6 3))
    (* 3 4)
    )
2
12
```

LOOP

The loop facility may be used for virtually all iterations required. Due to the versatility of the loop statement, there are many control parameters that are used to perform required tasks. These control parameters are discussed according to their function.

Iteration Control

The following iteration controls must precede other loop arguments except for the *with*, *initially*, and *finally* arguments. There can be any number of iteration controls in a single loop statement. The iterations occur simultaneously and the loop will finish when any one of the iterations is completed. See *loop* in AML Reference Manual for more details.

for

The for argument is a general increment (decrement) control. There are a number of paramenters that can be used to control how iteration should procede. The parameters are used with the loop's for control to accomplish many different types of iteration.

Examples:

```
AML> (loop for i from 0 to 3
        do (print i))
0
1
2
3
NIL
```

See Also:

```
FROM, TO, DOWNTO, UPTO
DOWNFROM, UPFROM
BY
=
THEN
IN
```

VALUE ACCUMULATION

The resulting accumulated value will be returned by the loop statement. When more than one accumulation is needed in a single loop statement the into parameter is needed to create a local variable to hold the results of the accumulations. If these accumulated values need to be returned, use the *finally* and *return* statements. See *loop* in AML Reference Manual for more details.

sum

The *sum* argument accumulates a running total of its parameter

Examples:

```
AML> (loop for i from 1 to 10
        sum i)
55
AML> (loop for i in '(2 4 5 6 8 9)
        collect (+ i 2)
        )
(4 6 7 8 10 11)
```

See Also:

```
append
collect
count
maximize
minimize
```

Conditional Execution

when

The *when* argument is used to perform some operations when a condition is true.

Examples:

```
AML> (loop for i from 0 to 10 by 3
        when (evenp i)
        do (print i)
        collect i)
0
6
(0 3 6 9)
AML> (loop for i from 0 to 10 by 3
        when (evenp i)
        do (print i) and
        collect i)
0
6
(0 6)
AML>
```

See Also:

```
loop (more details)
```

THE Referencing with *:from* keyword

The *the* reference construct has several keywords to augment its functionality. One keyword introduced here is *:from* which sets the starting point instance of a *the* reference to the specified object or property given after the *:from* keyword. This argument must evaluate to an object or property. Notice that all keywords in a *the* reference are given inside one set of parentheses to distinguish the end of the instance traversal path and the start of the keywords.

Format:

(the name-1 name-2 ... name-n (:from (the name-1 name-2 … name-m)))

Examples:

Assuming a tree structure as follows …

```
AIRPLANE                        [Level 1]
    maximum-speed               [Level 2]
    wing-span                   [Level 2]
    number-of-engines           [Level 2]
    WINGS                       [Level 2]
        WING-0001               [Level 3]
            span                [Level 4]
        WING-0002               [Level 3]
            span                [Level 4]
            RIBS                [Level 4]
                RIB-0001        [Level 5]
                    length      [Level 6]
                    width       [Level 6]
```

```
AML> (the airplane wings wing-0001 span)
40.1
AML> (the span (:from (the airplane wings wing-0001)))
40.1
AML> (children (the airplane wings))
(#<WING-CLASS #x1A8FDB8> #<WING-CLASS #x1AC32D8>)
AML> (loop
        for wing-kid in (children (the airplane wings))
        sum (the span (:from wing-kid))
        )
80.2
```

✓ Notice that the *wing-kid* local variable is used within the *loop* construct. The *wing-kid* variable evaluates to an instance of a wing class so that a span can be queried from it.

Exercise 8

After a close review of the class definitions and AML constructs described in the previous section, create a class definition for a mass-properties-class inheriting from *material-properties-class*. The mass-properties-class is inherited into the missile-body-component-class. The mass-properties-class assumes the *mass-source-object* is an object of dimension 2 (surface). The mass-properties-class has the following properties:

- *mass-source-object* which supplies a geometric object to all of its innate properties. This property is set dynamically at instantiation.
- *surface-area* which calls the method *volume-of-object* on the *mass-source-object*,
- *mass* which is calculated by multiplying the *material-thickness*, the *material-density* and the *surface-area* together. Add an "error check" using the *when* construct to ensure that the *mass-source-object* property is not *nil*.

Augment the classes inheriting from *missile-body-component-class* to set the formula of the *mass-source-object* property to be *!superior*. This can be implemented in the *missile-body-component-class* and will be inherited into its sub-classes.

Augment the *missile-geometry-class* by adding two output properties:

- *missile-body-components* which contains a list of subobjects of class *missile-body-component-class*,
- *missile-body-mass* which loops through the *missile-body-components* and sums their mass.

Exercise 8 Solution

```
(in-package :aml)

(define-class material-properties-class
  :inherit-from (object)
  :properties (
               ;;English units assumed
               material-thickness     (default 0.01)        ;ft
               material-name          (default 'aluminum)   ;symbol
               material-density       (default 165.0)       ;lb/ft^3
               )
  )

(define-class mass-properties-class
  :inherit-from (material-properties-class)
  :properties (
               mass-source-object nil
               ;;volume-of-object assumed to be only surface area here
               surface-area  (when ^mass-source-object
                                (volume-of-object ^mass-source-object)
                              )
               mass          (when ^surface-area
                                  (* ^surface-area
                                     ^material-density
                                     ^material-thickness
                                     )
                                  )
               )
  )

(define-class missile-body-component-class
  :inherit-from (mass-properties-class)
  :properties (
               ;;assumes this class will be mixed with a geometric class
               mass-source-object    !superior
               )
  )

(define-class spherical-nose-class
  :inherit-from (difference-object missile-body-component-class)
  :properties (
               ;;user defined properties
               missile-nose-radius (default 1.0)

               ;;internal properties
               object-list (list ^complete-sphere ^missile-body-object)
               (complete-sphere :class 'sphere-object
                diameter    (* ^^missile-nose-radius 2.0)
                solid?      nil
                )
               (missile-body-object :class 'cylinder-object
                ;;* 2.0 ensures the cylinder height is large enough
                height      (* ^^missile-nose-radius 2.0)
                diameter    (* ^^missile-nose-radius 2.0)
                orientation (list
                             (rotate 90 '(0 1 0))
                             (translate (list (/ ^height 2.0) 0 0))
                             )
                )
               )
  )

(define-class open-conical-nose-class
  :inherit-from (open-cone-object missile-body-component-class)
```

```
  :properties (
               ;;user defined properties
               missile-nose-length (default 2.0)
               missile-nose-radius (default 1.0)

               ;;internal properties
               height              ^missile-nose-length
               diameter            (* ^missile-nose-radius 2.0)
               orientation         (list
                                    (rotate -90 '(0 1 0))
                                    )
               )
  )

(define-class open-cylindrical-body-class
  :inherit-from (open-cylinder-object missile-body-component-class)
  :properties (

               )
  )

(define-class open-truncated-cone-body-class
  :inherit-from (open-truncated-cone-object missile-body-component-class)
  :properties (

               )
  )

(define-class hexagonal-profile-class
  :inherit-from (polygon-object)
  :properties (
               ;;user defined properties
               width     (default 1.0)
               height    (default 0.2)

               ;;internal properties
               vertices  (let* (
                                (half-w   (/ ^width 2.0))
                                (half-h   (/ ^height 2.0))
                                (sixth-w  (/ ^width 6.0))
                                )
                           (list
                            (list half-w 0.0 0.0)
                            (list sixth-w half-h 0.0)
                            (list (- sixth-w) half-h 0.0)
                            (list (- half-w) 0.0 0.0)
                            (list (- sixth-w) (- half-h) 0.0)
                            (list sixth-w (- half-h) 0.0)
                            ;;last point will be closed automatically
                            )
                           )
                )
  )

(define-class fin-profile-class
  :inherit-from (hexagonal-profile-class)
  :properties (
               ;;user defined properties
               fin-cord          (default 1.0)
               fin-thickness     (default 0.2)

               ;;internal properties
               width             ^fin-cord
               height            ^fin-thickness
               )
  )
```

```
(define-class fin-class
  :inherit-from (capped-surface-object)
  :properties (
               ;;user defined properties
               fin-cord          (default 1.0)
               fin-thickness     (default 0.2)
               fin-span          (default 3.0)

               ;;internal properties
               solid?            nil
               source-object     ^fin-extrusion

               (fin-profile :class 'fin-profile-class

                )

               (fin-extrusion :class 'extrusion-object
                swept-object      ^^fin-profile
                distance          ^^fin-span
                vector            '(0.0 0.0 1.0)
                )
               )
  )

(define-class fin-array-class
  :inherit-from (circular-array-object)
  :properties (
               ;;user defined properties
               fin-cord               (default 1.0)
               fin-thickness          (default 0.2)
               fin-span               (default 3.0)
               fin-span-offset        (default 2.0)
               fin-quantity           (default 4)

               ;;internal properties
               source-object      ^fin-source
               diameter           (* 2.0 ^fin-span-offset)
               quantity           ^fin-quantity
               ref-point          '(0.0 0.0 0.0)
               center             '(0.0 0.0 0.0)
               rotate-axis        '(0.0 0.0 1.0)

               (fin-source :class 'fin-class
                 orientation  (list
                               (rotate 90.0 '(0 1 0))
                               )
                )
               )
  )

(define-class missile-geometry-class
  :inherit-from (object)
  :properties (
        ;;user defined properties
        missile-general-body-radius      (default 1.0)
        missile-general-body-thickness   (default 0.01)

        missile-nose-length   (default 2.0)
        missile-nose-radius   (default ^missile-general-body-radius)
        missile-nose-type     (default 'sphere)
                              ;; options are 'sphere or 'cone

        missile-mid-body-length (default 14.0)
        missile-mid-body-radius (default ^missile-general-body-radius)

        missile-aft-body-length       (default 2.0)
        missile-aft-body-start-radius (default ^missile-general-body-radius)
```

```
        missile-aft-body-end-radius   (default 0.75)

        fin-distance-along-axis       (default 12.0)
        fin-cord                      (default 1.0)
        fin-thickness                 (default 0.2)
        fin-span                      (default 2.0)
        fin-span-offset               (default ^missile-general-body-radius)
        fin-quantity                  (default 4)

        display-coord-systems?        (default nil)

        ;;output
        missile-body-components (children !superior
                                     :class 'missile-body-component-class)
        missile-body-mass       (loop
                                     for kid in ^missile-body-components
                                     sum (the mass (:from kid))
                                     )
        )
:subobjects (
              (missile-coordinate-system :class 'coordinate-system-class
               display? ^^display-coord-systems?
               origin (list 0.0 0.0 0.0)
               )
              (nose-coordinate-system :class 'coordinate-system-class
                display? ^^display-coord-systems?
                origin     (if (equal ^^missile-nose-type 'cone)
                               (list (* 0.5 ^^missile-nose-length) 0.0 0.0)
                               (list ^^missile-nose-radius 0.0 0.0)
                               )
                reference-coordinate-system ^^missile-coordinate-system
               )
              (nose :class (case !missile-nose-type
                             (sphere   'spherical-nose-class)
                             (cone     'open-conical-nose-class)
                             (t        'spherical-nose-class)
                             )
                material-thickness  ^^missile-general-body-thickness
                reference-coordinate-system ^^nose-coordinate-system
               )
              (mid-body-coordinate-system :class 'coordinate-system-class
               display? ^^display-coord-systems?
               origin  (if (equal ^^missile-nose-type 'cone)
                           (list (+ (/ ^^missile-nose-length 2.0)
                                    (/ ^^missile-mid-body-length 2.0)
                                    )
                                 0.0
                                 0.0)
                           (list (/ ^^missile-mid-body-length 2.0) 0.0 0.0)
                           )
               reference-coordinate-system ^^nose-coordinate-system
               )
              (mid-body :class 'open-cylindrical-body-class
                diameter       (* ^^missile-mid-body-radius 2.0)
                height         ^^missile-mid-body-length
                material-thickness  ^^missile-general-body-thickness
                orientation    (list
                                   (rotate 90 '(0 1 0))
                                   )
                reference-coordinate-system ^^mid-body-coordinate-system
                )
              (aft-body-coordinate-system :class 'coordinate-system-class
                display? ^^display-coord-systems?
                origin     (list
                               (+ (/ ^^missile-mid-body-length 2.0)
                                  (/ ^^missile-aft-body-length 2.0)
                                   )
```

```
                                0.0
                                0.0)
                  reference-coordinate-system ^^mid-body-coordinate-system
                 )
                (aft-body :class 'open-truncated-cone-body-class
                  start-diameter     (* ^^missile-aft-body-start-radius 2.0)
                  end-diameter       (* ^^missile-aft-body-end-radius 2.0)
                  height             ^^missile-aft-body-length
                  material-thickness ^^missile-general-body-thickness
                  orientation        (list
                                         (rotate 90 '(0 1 0))
                                         )
                  reference-coordinate-system ^^aft-body-coordinate-system
                  )
                (fins :class 'fin-array-class
                  orientation  (list
                                (rotate 90 '(0 1 0))
                                (translate (list ^^fin-distance-along-axis
                                                 0.0
                                                 0.0)
                                    )
                                )
                  reference-coordinate-system ^^missile-coordinate-system
                 )
                )
    )
```

Code Explanation

The developer creates a generic *mass-properties-class* and inherits it into the *missile-body-component-class*. This also promotes code re-use and modular design as the generic class can be instantiated by itself or inherited into other user-defined classes in the future. This class is slightly different than the *material-properties-class* because of the *mass-source-object*. This allows a user to instantiate this class independently from any object and "point" to an other object by setting the *mass-source-object* property dynamically at instantiation or at runtime. The other properties in the *mass-properties-class* contain a *when* statement before their main execution in order to trap errors. Notice that the method *volume-of-object* is expected to return a surface area in this particular class. This could be augmented very easily with a *case* statement or a series of *if* statements.

Notice how the *missile-body-component-class* helps in this situation. With the simple addition of the *mass-properties-class*, all of the missile components receive the functionality from the *mass-properties-class*. The *mass-source-object* could also be set at each instance of the *missile-body-component-class*. This is a developer decision that may be considered more of a matter of "style" rather than modeling "correctness".

The *missile-geometry-class* now has the two output properties. The *missile-body-components* property could be referred to in other objects or properties that need these specific objects. This becomes more efficient if these objects are needed in several places because a *the* reference will be used to obtain the objects instead of making an other call to the *children* method. The *select-object* function could also be used here if some desired objects are located in objects within the children.

Note that the code for the *missile-body-mass* could have simply been given as follows:

```
(+  (the mass (:from ^nose))
    (the mass (:from ^mid-body))
    (the mass (:from ^aft-body)))
```

This is certainly simpler than the loop given in the exercise, however it is extremely limited. For example, with the simple addition of the masses by direct *the* references, the missile component names can not change without changing the code. Also, the code would have to be changed any time the a user adds a new body component to be included in the mass calculation. The use of *children* satisfies all of these concerns because it will always have a dependency on the missile's children. Each time a child is added or deleted, the *children* method will make the *missile-body-mass* become unbound. This practice of using the methods such as *children* or functions like *select-object* enables the program to become dynamic/adaptive and allow a user to dynamically create a tree structure with properties and formulas that automatically update based on demand.

3. Introduction to AML Graphical User Interface (GUI) Design

3.1 Preview

Using AML, the application developer can build graphical user interfaces using AML classes, functions and tools. Building a user interface in AML can be done at different levels. If the developer is looking for maximum design flexibility and control, the GUI development procedure can be at the base level, i.e. GUI components are built one by one and linked to the application model through user-defined methods and functions. This process requires the most development time. This functionality is documented in the AML Model Interface Design Manual in the chapter titled "Base Classes and Tools".

On the other hand, GUI development can be at a much higher level if the developer decides to use a set of standard AML property classes that are capable of automatically interfacing an AML model providing most of the standard control that a user requires from an application. This reduces development time dramatically. This functionality is documented in the AML Model Interface Design Manual in the chapter titled "Model Interface Classes and Techniques". Mixing functionality and objects from both of these sections is allowed and also recommended for some applications.

3.2 Automated Model Interface Design

This section introduces a portion of the automatic GUI generation capabilities. Using the AML GUI classes and functionality, the developer has ultimate control over the functionality and the aesthetics of the GUI as well as the communication between the GUI and the model. For large applications however, this may be undesirable due to the fact that the developer has more code to design and debug. If the developer follows some standard design techniques and use some advanced application forms, the GUI-specific widgets, functions or GUI-model communication are done automatically with the AML model interface property classes and some of the advanced GUI forms available within AML. Developed using standard AML base GUI classes, the model interface system revolves around the following idea: the developer builds a model with enough knowledge in its objects and properties (Property nature, behavior and rules) so that advanced GUI classes can provide all the functionality expected from a graphical user interface without requiring any programming. For that purpose, a set of property classes are given in AML that, when used in the context of what is defined as a *data-model*, contain enough knowledge to automatically generate their own user interface. It is suggested at this stage, the developer refers to the introduction section of the *Model Interface Design* manual for further reference.

The concept of a *data-model* is introduced here to aid in the user interface design and general AML model organization*:* The part of an AML model typically represented by a set of properties that are meant to be available to the user of an AML application usually through a graphical user interface. These properties are typically instances of *model-interface-property-class* introduced below. These classes implement the expanded property specification mentioned earlier in the manual.

Model-interface-property-class **[Class]**

This is the super-class of all *model-interface* property classes. This class is typically not instantiated by the user.

Inherit-from: property-object

Properties:

Available?	Flag (*t* or *nil*) specifying if the property is available/usable, usually based on the current values of the other *data-model* properties. A non-available property is grayed out by the GUI.
Label	String specifying the label of the property. The label is typically used by the GUI. Default formula is *(write-to-string (object-name (the superior))).*

✓ Note that the *formula* slot for any class inheriting from *model-interface-property-class* must be specified otherwise the *formula* will be set to *nil*. In order to keep the original formula from the inherited class, set the *formula* to *:inherit-formula*.

Computed-data-property-class **[Class]**

This class is used for general-purpose properties whose values are not supposed to be edited by the user. These properties are typically output properties.

Inherit-from: model-interface-property-class

Editable-data-property-class **[Class]**

This class is used for general-purpose properties whose values are to be edited by the user. These properties are typically input properties.

Inherit-from: model-interface-property-class

Flag-property-class **[Class]**

This class is used for "flag" properties that expect a value of t or nil. These properties are typically represented in the GUI with a toggle button.

Inherit-from: model-interface-property-class

Option-property-class **[Class]**

This class is used for properties whose *value* is equal to one element of a list of available options. These properties can be represented with an option menu, a radio button group, or a combo box. A combo box is typically used whenever the *value* of the *option-property-class* is allowed to be different for any of the available options.

Inherit-from: model-interface-property-class

Properties:

Labels-list	List of strings specifying a label for each option. *Label-list* is ignored when *mode* is *'combo* because a user is allowed to edit an option making options and labels one entity.
Mode	GUI mode specifying the GUI representation of the property. Allowed values are *'radio,' menu,* and *'combo* (See Section 3 of the Model Interface Design Manual for corresponding GUI Widgets). Default formula is *'radio.*
Options-list	List of available options.

Data-model-node-mixin **[Class]**

This class is designed to represent a node of a data model. It contains application-specific knowledge about the node it represents. It also manages the properties of that node that are part of the *data-model* and hence that should be a part of the application GUI. *Data-model-node-mixin* is typically inherited into user defined classes. Standard application forms (described in the Model Interface Design Manual) make use of data model nodes for the automatic generation of model GUIs. Application forms that contain *data-model* in their class name are typically designed to interface *data-model-node-mixin*s.

Inherit-from: *object*

Properties:

Available?	Default formula is *(Default t).* When nil, does not allow GUI access to the *property-object-list* of the node from the data model tree.
Label:	Label of the node as displayed in the GUI data model tree.
Property-objects-list	List of *model-interface-property* instances that define the *data-model* properties of this node. Properties included in this list will be available in the automatically generated forms that use *data-model-node-mixin*. The list follows the same format of the *property-objects-list* of *ui-multiple-property-subform-class*.

See also: ui-multiple-property-subform-class, ui-data-model-tree-inspection-class, ui-data-model-main-form-class.

Example:

Below is the class definition of a box, a cylinder and their union. They can be instantiated as a *data-model* that one of the **application forms** defined later can point to. Figure 19 shows the end result.

```
(define-class box-data-model-class
  :inherit-from(box-object data-model-node-mixin)
  :properties(
          property-objects-list
            (list
                (list (the superior depth self) '(automatic-apply? t))
                (the superior height self)
                (the superior width self)
                )
          )
  )

(define-class cylinder-data-model-class
  :inherit-from(cylinder-object data-model-node-mixin)
  :properties(
          property-objects-list
            (list
                (the superior diameter self)
                (the superior height self)
                )
          diameter 1.0
          height   8.0
          )
  )

(define-class union-data-model-class
  :inherit-from(union-object data-model-node-mixin)
  :properties(
          label "Union of Box and Cylinder"
          property-objects-list
            (list
                (the superior simplify? self)
                (the superior render self)
                )
          (simplify? :class 'flag-property-class
             formula t
             )
          (render :class 'option-property-class
              options-list '(boundary shaded facet)
              mode 'menu
              formula :inherit-formula
              )
          object-list (list ^box ^cylinder)
          )
  :subobjects(
          (cylinder :class 'cylinder-data-model-class
            label "Cylinder node"
            )
          (box :class 'box-data-model-class
           label "Box Node"
           )
    )
)
```

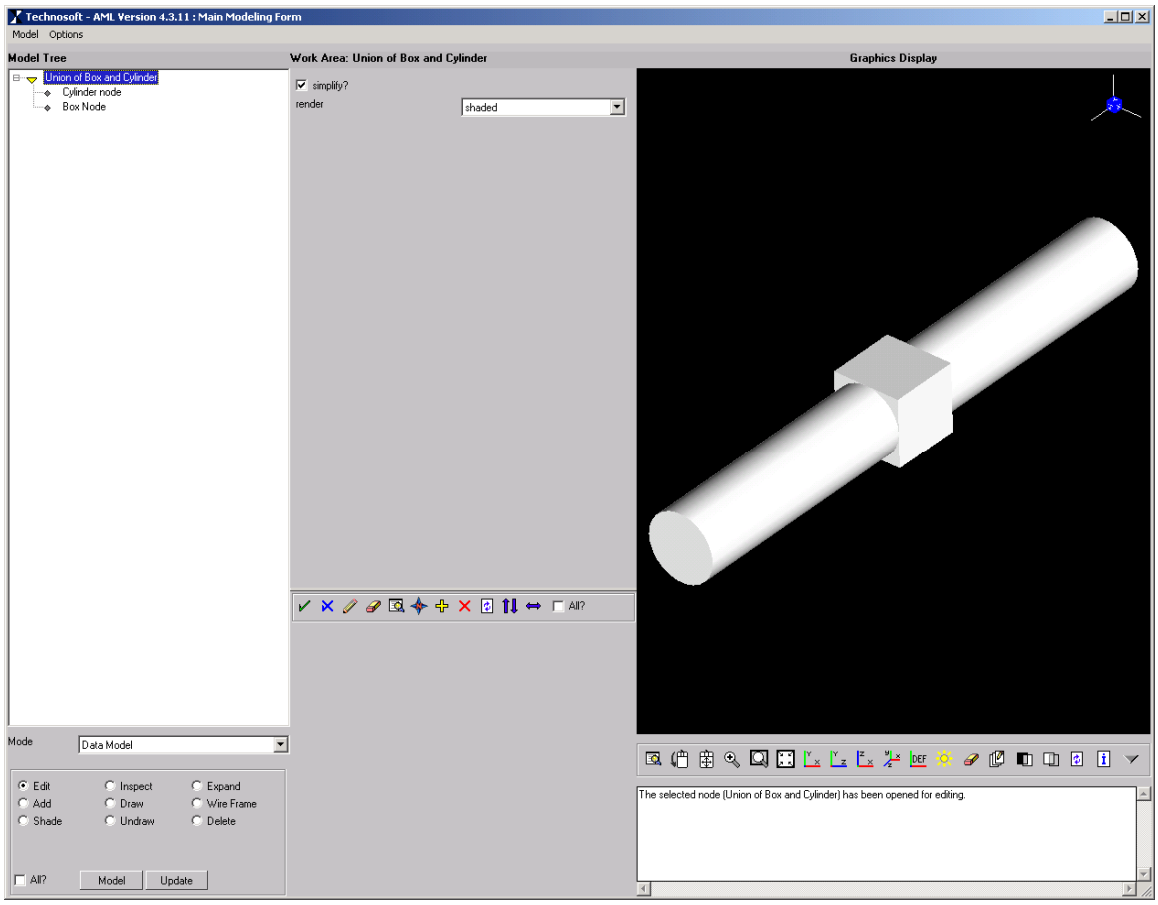


Figure 19

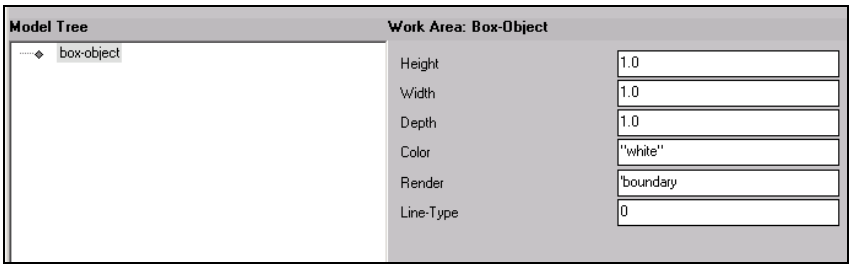
property-names-to-inspect **[Method]**

This method is automatically called by the system when an object is opened for inspection or editing if no property-objects are defined in the *property-objects-list*. The AML developer can redefine this method on any class to specify the properties that are visible when an instance of that class is opened for "inspection" (Given that the "Show all" toggle on the object inspection form is off). This method should always return a list of property names. This method is pre-defined on a number of AML classes, it returns nil for all other classes. When it returns nil, all properties will be visible. The "Show all" toggle button can always be toggled on to show all properties on the inspection form.

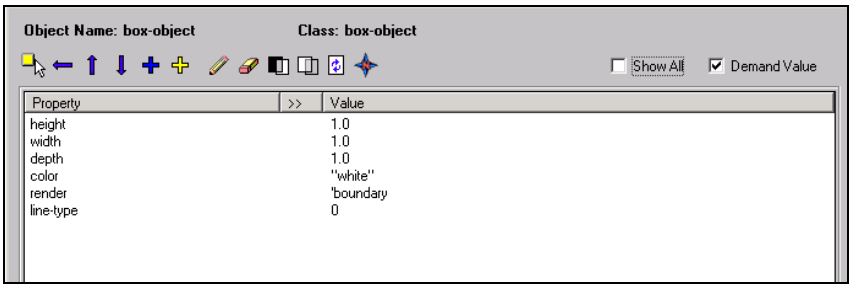
Example:

```
(define-method property-names-to-inspect box-object ()
  '(height width depth color render line-type)
  )
```

Upon inspecting an instance of a *box-object*, the inspect form will only show the properties defined in this method. Similar results occur when editing an instance of a *box-object* as seen in the following figures.



Editing an instance of a *box-object* with the *property-names-to-inspect* method redefined.



Inspecting an instance of a *box-object* with the *property-names-to-inspect* method redefined.

Exercise 9

After a close review of the class definitions and AML constructs described in the previous section, create a class definition for a data model called *missile-geometry-data-model-class* inheriting from *data-model-node-mixin* and *missile-geometry-class*. Overwrite the properties for the needed properties to create the form shown in Figure 20. Use the *:inherit-formula* specification for each of the properties' formula to allow the base class' formula to be used. All of the properties shown are instances of *editable-data-property-class* with the exception of the *missile-nose-type* (class *option-property-class*), *missile-body-mass* (class *computed-data-property-class*), and *display-coord-systems?* which is an instance of a *flag-property-class*. Make the *missile-nose-length* available only if the *missile-nose-type* is a conical nose. The *missile-nose-type* should use the *automatic-apply?* property in the *init-form* portion of the *property-objects-list* so that the *missile-nose-length* property updates as the user interactively modifies *missile-nose-type* without pressing the Apply button. The *missile-body-mass* should use the *automatic-demand?* property in the *init-form* portion of the *property-objects-list* so that the *missile-body-mass* property does not demand the value of the *missile-body-mass* without the user clicking on the "!" button. Also separate the properties into their respective functional areas by using text strings in the *property-objects-list* as shown in Figure 20.

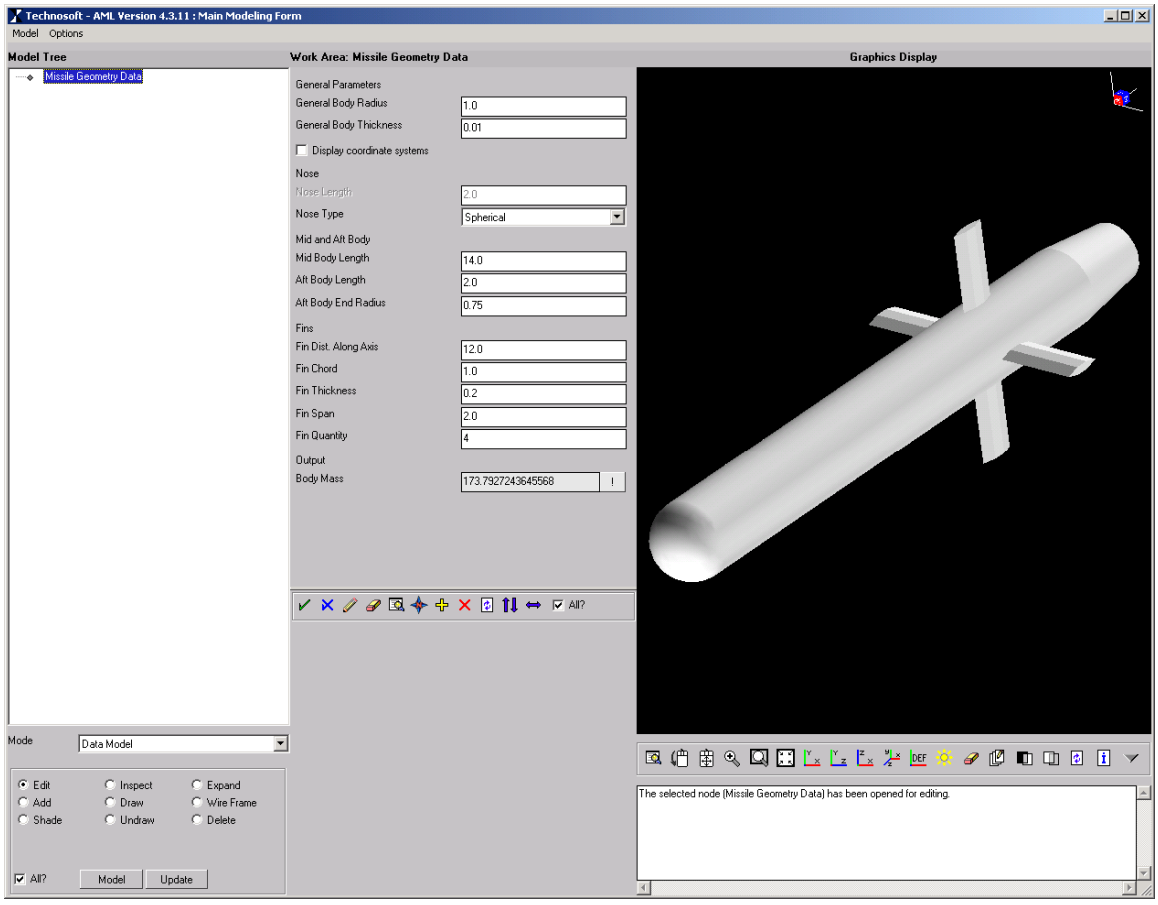


Figure 20

Exercise 9 Solution

```
(define-class missile-geometry-data-model-class
  :inherit-from (missile-geometry-class data-model-node-mixin)
  :properties (
               property-objects-list
               (list
                "General Parameters"
                (the superior missile-general-body-radius self)
                (the superior missile-general-body-thickness self)
                (the superior display-coord-systems? self)
                "Nose"
                (the superior missile-nose-length self)
                (list
                  (the superior missile-nose-type self)
                  '(automatic-apply? t)
                  )
                "Mid and Aft Body"
                (the superior missile-mid-body-length self)
                (the superior missile-aft-body-length self)
                (the superior missile-aft-body-end-radius self)
                "Fins"
                (the superior fin-distance-along-axis self)
                (the superior fin-cord self)
                (the superior fin-thickness self)
                (the superior fin-span self)
                (the superior fin-quantity self)
                "Output"
                (list
                  (the superior missile-body-mass self)
                  '(automatic-demand? nil)
                  )
                )
         label  "Missile Geometry Data"

        (missile-general-body-radius :class 'editable-data-property-class
                label "General Body Radius"
                formula :inherit-formula
                )
        (missile-general-body-thickness :class 'editable-data-property-class
                label "General Body Thickness"
                formula :inherit-formula
                )
        (missile-nose-length :class 'editable-data-property-class
                label      "Nose Length"
                formula    :inherit-formula
                available? (equal ^missile-nose-type 'cone)
                )
        (missile-nose-type :class 'option-property-class
                label "Nose Type"
                mode 'menu
                formula :inherit-formula
                options-list '(sphere cone)
                labels-list '("Spherical" "Conical")
                )
        (missile-mid-body-length :class 'editable-data-property-class
                label "Mid Body Length"
                formula :inherit-formula
                )
        (missile-aft-body-length :class 'editable-data-property-class
                label "Aft Body Length"
                formula :inherit-formula
                )
        (missile-aft-body-end-radius :class 'editable-data-property-class
```

```
                label "Aft Body End Radius"
                formula :inherit-formula
                )
        (fin-distance-along-axis :class 'editable-data-property-class
                label "Fin Dist. Along Axis"
                formula :inherit-formula
                )
        (fin-cord :class 'editable-data-property-class
                label "Fin Chord"
                formula :inherit-formula
                )
        (fin-thickness :class 'editable-data-property-class
                label "Fin Thickness"
                formula :inherit-formula
                )
        (fin-span :class 'editable-data-property-class
                label "Fin Span"
                formula :inherit-formula
                )
        (fin-quantity :class 'editable-data-property-class
                label "Fin Quantity"
                formula :inherit-formula
                )
        (missile-body-mass :class 'computed-data-property-class
                label "Body Mass"
                formula :inherit-formula
                 )
        (display-coord-systems? :class 'flag-property-class
                label "Display coordinate systems"
                formula :inherit-formula
                )
               )
)
```

Code Explanation

The *missile-geometry-data-model-class* had all of the behavior of the *missile-geometry-class* with the additional ability to show these properties in an automatically generated user interface. A user can now modify these properties easily by pressing the *apply* button on the *data model form*. Notice how the intelligence is placed in the model so that the *missile-nose-length* is not available when the *missile-nose-type* is spherical. The *apply* button knows to not apply any changes to unavailable buttons and apply changes (if any were made) to the available buttons. Notice the order of the *property-objects-list* which corresponds to the order that the buttons are instantiated from top to bottom.

Notice the use of *:inherit-formula* within all of the data properties. This allows for the specification of the formula in the original class that is inherited into the data model representation. Formulas can be overwritten in the data model, however it is not required, and often is not needed because of the *:inherit-formula* functionality.

4. AML Source Code Management (AML Systems)

This section of the manual introduces the concepts and syntax of systems in AML. The management of source code is accomplished through the definition of systems. A system is a set of source code files that are grouped together. Defining a system allows the code in a system to be treated as a module that may be loaded, compiled, and archived as a single entity. Compiling systems archives the source code with the binary files so updating older versions is possible by using the archived code. A system also compiles binaries for multiple platforms within a version to allow different platforms to be operating with the same system version. A system may require other systems to automatically load before loading or compiling itself. Organizing code into systems that may be loaded is a methodology for the reuse of code. A logical path is a reference to files and directories in the system. The logical path file stores the logical path references, making modification easy. These references are converted by the use of the *logical-path* function.

New AML Constructs

The following AML concepts and constructs are used in this section:

- logical path files
- *logical-path*
- *define-system*
- *compile-system*
- load-system

Logical Paths File

AML has the capability of defining logical-path-reference variables to locate resources on the file system. They are defined in logical paths files. On Unix platform, a user logical paths file *(logical.paths)* can be created under the user's home directory. On WINDOWS platforms, a logical paths file *(logical.pth)* exists under the AML directory. The user can append entries to this file and make modifications as necessary. The logical paths file contains lines with logical-path-reference and corresponding path definitions. These can be direct or relative as shown below.

```
:home                    ~/
:tmp                     /tmp/
:missile                 :home       missile/
```

Compiler directives can be used in a logical paths file and amidst AML source code to distinguish between a WINDOWS and a non-WINDOWS entry in the following way:

```
#-WINDOWS
:home                    ~/
#+WINDOWS
:home                    c:\users\user1\
```

A #+WINDOWS directive specifies that the following line is only visible on WINDOWS machines, while a #-WINDOWS directive specifies the the following line is only visible on non-WINDOWS machines. Directives should not be followed by logical path entries on the same line, and only one directive per line is allowed.

logical-path [Function]

logical-path is a function that uses the logical paths file to convert a logical-path-reference to a path definition. The path definition is a string that is retrieved from the logical paths file. If the logical path reference does not exist in the logical paths file, the function returns nil.

Format:

(logical-path logical-path-name [file-name-1 ... file-name-n])

Arguments:

logical-path-name	The name of a logical path reference defined in logical paths file.
file-name	A string that is concatenated to logical-path-name reference.

Examples:

On the Windows platform …

```
AML> (logical-path :home)
"c:\\users\\user1\\"
AML> (logical-path :temp)
"c:\\temp\\"
AML> (logical-path :temp "mesh-01" "nodes.dat")
"c:\\temp\\mesh-01\\nodes.dat"
```

On the UNIX platform …

```
AML> (logical-path :home)
"/users/user1/"
AML> (logical-path :temp)
"/tmp/"
AML> (logical-path :temp "mesh-01" "nodes.dat")
"/tmp/mesh-01/nodes.dat"
```

Note: On the Windows platform, file paths must have double slashes. The *logical-path* function takes care of this for the user.

DEFINE-SYSTEM [Function]

The *define-system* construct is the main mechanism for creating systems. The definition should appear in a system definition file named *system.def*. The *system.def* must be in the directory returned when *(logical-path :system-name)* is executed. The system-name must be an entry in the logical path file for the system to be found for compiling or loading.

Format:

(define-system system-name [:require-systems '()] [:files '()])

Arguments:

system-name — The name of the system being defined. It is recommended that the system have a keyword name to make it package independent.

require-systems — A list of system names that must be loaded before the system being defined may be compiled or loaded.

files — A list of files that comprise the system. The files may be located in the same directory as the system.def file or in a subdirectory named sources only.

Example:

```
(define-system :MY-SYSTEM
:require-systems '(:base-system :extension-system)
:files '(
        "file1.aml"
        "file2.aml"
        "file3.aml"
        )
)
```

A directory structure for a system, named *my-system*, would look something like the following:

```
/home/
    apps/
        my-system/
            system.def ;; File containing the system definition.
            sources/
                file1.aml ;; Source code file.
                file2.aml ;; Source code file.
                file3.aml ;; Source code file.
```

In this example, the logical path file entry for *my-system* must be:

```
:my-system      /home/apps/my-system/
```

compile-system [Function]

When a system is compiled the source code is read from the files specified in *define-system*. Compilation of a system will create system versions (archives) that contain the source from time of compile and the binary files created by those source files. Binary files are created in a subdirectory named for the machine type in the system version subdirectory. That subdirectory will be used by *load-system* to load the binary files. A system tracks the binary files created and will not compile source files that have not changed since the last being compiled unless the *force?* keyword is t. Only the newest version or a new version may be compiled.

Format:

(COMPILE-SYSTEM system-name [:force? nil] [:forget? nil] [:new? nil])

Arguments:

system-name	The name of a system to compile. The system name must be an entry that is found in the logical.paths file.
force?	Default is nil which only compiles files that have changed since the last compile. A t value will compile all files in the system.
forget?	This keyword controls the redefinition of a system. The default is nil which will not read the system.def file after the first compile or load. If *forget?* is supplied as t the system.def file will be reread. Note: Changes made to a system.def file of a loaded system will not be recognized unless *forget?* is t.
new?	Defaults to nil. When t, creates a new system archive. If *new?* is nil the system will be archived and compiled into the newest system archive.

Examples:

```
; The following compiles the system files the first time on a Sun
; machine and creates MY-SYSTEM-1 subdirectory structure shown below.
AML> (compile-system :my-system)
/home/
    apps/
        my-system/
            system.def File containing the system definition.
            sources/
                file1.aml Source code file.
                file2.aml Source code file.
                file3.aml Source code file.
            MY-SYSTEM-1/
                system.def
                sources/
                    file1.aml Archived source code file.
                    file2.aml Archived source code file.
                    file3.aml Archived source code file.
                SUN-bins-a/
                    file1.sbin Archived binary file.
                    file2.sbin Archived binary file.
                    file3.sbin Archived binary file.
; The following reads the system.def file and compiles only changed
; or uncompiled files into the existing version.
AML> (compile-system :my-system :forget? t :force? nil :new? nil)
```

load-system [Function]

When a system is loaded the binary files that were created during the last compilation are loaded if no version number is supplied. When a version number is supplied the binaries for the machine will be loaded from a compile that may not be the newest. This allows versions to be in production and newer versions to be under development. If the source code is changed the changes will not be loaded until after a *compile-system* is performed. A system also tracks the version of the binary files that are loaded so that successive loading of the same system will not take time to load files that are unchanged.

Format:

(LOAD-SYSTEM system-name [:forget? nil] [:version nil])

Arguments:

system-name	The name of a system to load. The system name must be an entry that is found in the logical path file.
forget?	This keyword controls the redefinition of a system. The default is *nil* which will not read the system.def file after the first compile or load. If *forget?* is supplied as *t* the system.def file will be read.
version	This keyword specifies the version of the binaries which will be loaded. The default value of *nil* will load the latest version.

Note: Changes made to a system.def file of a loaded system will not be recognized unless *forget?* is *t*.

Example:

```
AML> (load-system :my-system)
;;; Loading source file "/home/apps/system/MY-SYSTEM-1/sysdef.def"
Loading system MY-SYSTEM...
;;; Loading binary file "/home/apps/system/MY-SYSTEM-1/SUN-bins-a/
file1.sbin"
;;; Loading binary file "/home/apps/system/ MY-SYSTEM-1/SUN-bins-a/
file2.sbin"
;;; Loading binary file "/home/apps/system/ MY-SYSTEM-1/SUN-bins-a/
file3.sbin"
MY-SYSTEM System loaded.
;; Loading again will not reload files that have not be recompiled.
AML> (load-system :my-system)
Loading system MY-SYSTEM...
Skip loading: /home/apps/system/MY-SYSTEM-1/SUN-bins-a/file1.sbin
already loaded.
Skip loading: /home/apps/system/MY-SYSTEM-1/SUN-bins-a/file2.sbin
already loaded.
Skip loading: /home/apps/system/MY-SYSTEM-1/SUN-bins-a/file3.sbin
already loaded.
MY-SYSTEM System loaded.
```

See Also:

```
compile-system-file
```

Exercise 10

Take the code from the previous exercise and divide it into functional groupings of classes and place them into different files based on their grouping. Define a system called *:missile-training-system* that requires the files from the functional groupings. Modify your logical path file to have a corresponding entry for the location of the system. Compile this system, exit from AML and practice loading the system. You do not have to exit, but it demonstrates the ease loading a system from scratch and having all of the class definitions in memory.

The following table shows a suggested functional grouping of the classes from the previous exercise. The order of the files in the *system.def* file should follow the order given in the table.

Classes	**Filename**
• *material-properties-class* • *mass-properties-class*	base-functionality.aml
• *hexagonal-profile-class* • *fin-profile-class* • *fin-class* • *fin-array-class*	fins.aml
• *missile-body-component-class* • *spherical-nose-class* • *open-conical-nose-class* • *open-cylindrical-body-class* • *open-truncated-cone-body-class*	missile-body-components.aml
• *missile-geometry-class*	missile-geometry.aml
• *missile-geometry-data-model-class*	missile-interface.aml

Exercise 10 Solution

```
(in-package :aml)

(define-system :missile-training-system
  :files '(
           "base-functionality.aml"
           "fins.aml"
           "missile-body-components.aml"
           "missile-geometry.aml"
           “missile-interface.aml”
           )
  )
```

Code Explanation

The system loads/compiles the files in order from top to bottom. It is important to know the order of files because some files may contain items that require other classes/functions/methods to be previously loaded.

5. Defining Functions and Methods

This section of the manual presents some topics that are useful to a developer such as:

- defining functions with *defun*
- defining methods with *define-method*

5.1 Defining Functions

defun **[Construct]**

The *defun* construct is used to define a function, or procedure in AML.

Format:

(DEFUN function-name ([args]) [body])

Arguments:

function-name	The function name must be a symbol (given without a quote).
args	Arguments for the function.
body	The action(s) the function performs.

Examples:

```
(defun quadratic-formula (a b c)
   (let* (
          (radical      (- (expt b 2) (* 4 a c)))
          (denominator      (* 2 a))
          (numerator-plus   (+ (- b) (sqrt radical)))
          (numerator-minus  (- (- b) (sqrt radical)))
          )
      (list
       (/ numerator-plus denominator)
       (/ numerator-minus denominator))
      )
   )
AML> (quadratic-formula 1 3 2)
(-1.0f0 -2.0f0)
```

Notes:

There are more ways to use arguments than shown above such as “keywords” and “optional” arguments. Time permitting, these may be covered in the training class.

5.2 Defining Methods

define-method **[Construct]**

A method is an operation (function) that is defined specifically for a class. For example suppose the method *volume* is defined for each of the classes *box-object*, *cylinder-object*, and *sphere-object*. This method calculates the volume for an object. By calling the *volume* method with an instance of one of those classes, the correct operation will be executed automatically and the volume of the instance returned. Inheriting from a class that has methods defined for it will also inherit the methods. The following construct is used for defining methods in AML.

✓ *The* referencing is modified to start at the instance used as the first argument of the method call.

Notes: Do not use *self* as a variable within define-methods because *self* is locally bound by the system to be the instance used to call the method. Within a method defined on an object other than a property-object, *(the)* returns the object itself. Within a method defined on an instance of *property-object* (or anything that inherits from *property-object*), *(the)* returns the value of the property.

Format:

(DEFINE-METHOD method-name class ([args]) [body])

Arguments:

method-name	The method name must be a symbol (given without a quote).
class	The class for which the method is written.
args	Arguments for the method.
body	The action(s) the method performs.

Examples:

```
(define-class EXAMPLE-BOX-CLASS
  :inherit-from (box-object)
  :properties (
               height  3.5
               width   4.1
               depth   (* 1.8 ^height)

               box-weight  (weight !superior 0.3)
                )
  )

(define-method WEIGHT BOX-OBJECT (density)
    (let* (
            (volume (* (the height) (the width) (the depth)))
            )
```

```
          (* density volume)
        )
    )

(define-method WEIGHT CYLINDER-OBJECT (density)
    (let* (
            (volume (/ (* pi (the diameter) (the diameter) (the height))
                       4.0))
            )
          (* density volume)
        )
    )

AML> (create-model 'EXAMPLE-BOX-CLASS)
#<EXAMPLE-BOX-CLASS @ #x21a2975a>
AML> (the box-weight)
27.121499999999994
AML> (create-model 'box-object)
#<BOX-OBJECT @ #x223b2a62>
AML> (weight (the) 8.4)
8.4
AML> (create-model 'cylinder-object)
#<CYLINDER-OBJECT @ #x223c194a>
AML> (weight (the) 1.0)
1.5707963267948966

(define-method WEIGHT GRAPHIC-OBJECT (density)
    (let* (
           (volume (volume-of-object (the)))
           )
       (* density volume)))

(define-class INTERSECTION-EXAMPLE-CLASS
    :inherit-from (intersection-object)
    :properties (
        object-list (list ^part ^shaft)
        (part :class 'box-object
            solid? t
            )
        (shaft :class 'cone-object
            height 4.0
            diameter 2.0
            solid? t
            )
        )
    )

AML> (create-model 'intersection-example-class)
#<INTERSECTION-EXAMPLE #x19FFB44>
AML> (weight (the) 1.0)
0.744320226850907
```

✓ Notes:

- There are more ways to use arguments than shown above such as “keywords” and “optional” arguments. Time permitting, these may be covered in the training class.
- A define-method may not be defined for a class that has not been previously defined and loaded into memory. It is suggested that methods immediately follow the classes they are defined for.
- Define-methods are defined for classes but are executed by calling the method with an instance of the class for which it is defined .

- The most specific method for the instance will be used. Consider the example given for define-class. If a method has been defined for table-top (which inherits from *box-object*) which has the same name as one defined for *box-object*, the method on table-top will take precedence when called on an instance of table-top.

6. Low Level User Interface Design

This section introduces some basic low level AML user interface design techniques through an example and an exercise. It is useful to go first through some naming conventions:

1- Widget: A user interface entity (i.e. a button, a menu…) that is usually an instance of an AML class.

2- Form: A widget used to group other widgets, can be a stand-alone window or a subobject of another form.

3- *Top-level* Form: A form that is a stand-alone window, i.e. not a subobject of another form.

4- Component: A widget that is not used to group other widgets. It is typically a subobject of a form.

5- Layout: An object used to group *top-level* forms. *Top-level* forms are typically subobjects of a layout.

6- An instance of class *X* is an instance of *X* or of any class that inherits from X.

6.1.1 Positioning and Sizing

Below are the class definitions of *ui-widget, ui-group, ui-form-class*. Except for *ui-form-class*, these two classes should not be instantiated by the developer. *Ui-widget* is the super-class of all widget classes and *Ui-group* is the super-class of all form classes. For the purpose of this introductory section, the definitions below do not show all the properties of the classes described and focus only on positioning and sizing. Advanced sizing and positioning properties are also defined with the classes *ui-grid-form-mixin* and *ui-grid-component-mixin* (and their subclasses) but will not be covered in this section. Please refer to the *GUI Base Classes* Manual for complete reference.

UI-WIDGET **[Class]**

Ui-widget is the super-class of all widgets. A widget is a GUI entity that has the capability to control its own size/position and appearance attributes. Unless stated otherwise, all GUI widgets are instantiated as subobjects of a *ui-group* instance. All *ui-widget* instances (i.e. instances of classes that inherit from *ui-widget*) base their position and size on the properties *x-offset, y-offset, width, height* and *measurement*.

Inherit-from: ui-root

Properties:

Gray?: Defaults to nil. When t, the widget is disabled and appears grayed out.

Measurement: Defaults to *(default 'percentage).* Can take a value of 'pixels or 'percentage. When 'pixels, the *x-offset, y-offset, width* and *height* of the widget represents pixel values. When 'percentage, the *x-offset, y-offset, width* and *height* of the widget represents a percentage value of the width/height of the superior object of

the *ui-widget* instance. Also, when a percentage measurement is specified, the widget is attached to its parent, i.e. the widget grows and shrinks with its parent window when that window is resized with the mouse.

X-offset: Defaults to *0*. Integer representing the offset of the widget from the left side of its parent.

Y-offset: Defaults to *0*. Integer representing the offset of the widget from the top side of its parent.

Width: Defaults to *10*. Integer specifying the width of the widget.

Height: Defaults to *10*. Integer specifying the height of the widget.

UI-GROUP [Class]

Ui-group is the Superclass of all form classes. Typically, a class that *ui-group* inherits into is instantiated as a parent object to other widgets and can manage their position, size, appearance, property values and callbacks. A group can also be a subobject of another group.

Inherit-from: ui-widget

UI-FORM-CLASS [Class]

A *ui-form-class* is a *ui-group* that can be instantiated as a *top-level* form or as a subobject of another *ui-group*. A *Ui-form-class* is typically instantiated as a *top-level* form. When displayed, it is automatically created inside a stand-alone window with a border and a title bar. A *ui-form-class* instance, when a *top-level* form, ignores the value of its *measurement* property: Its *x-offset*, *y-offset*, *width* and *height* properties are pixel values. However, setting the measurement property of a *top-level* form is very useful since the measurement property value of its children widgets is directly derived from their parent form.

Inherit-from: ui-group

Note: For efficiency purposes, if the developer wishes to define a form that will only be instantiated as a subobject of another form, it is recommeded to use *ui-subform-class* instead of *ui-form-class*. Refer to the *GUI Base Classes* reference manual.

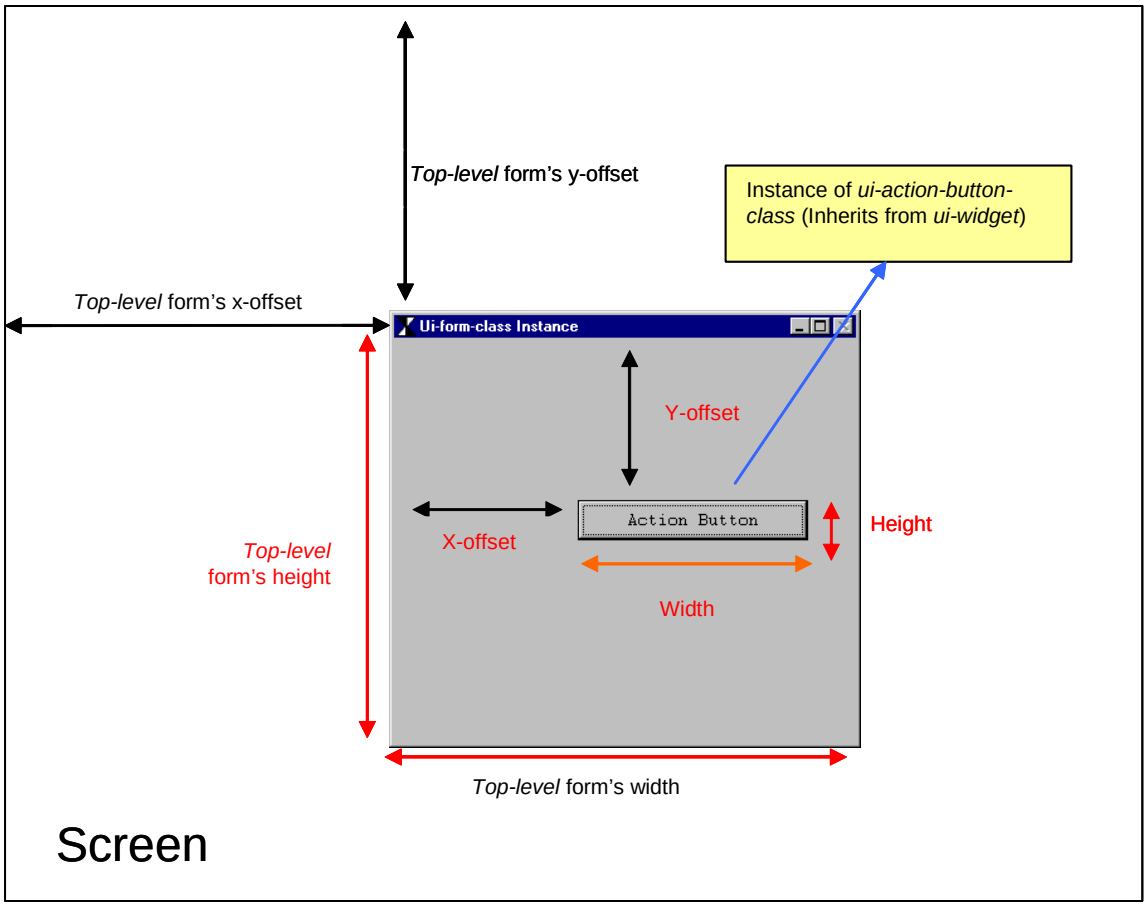


Figure 21

A *top-level* form is a parent (superior) to a group of components and other forms. Figure 21 shows a form with only one component. Below is the source code that defines its class and instantiates it as a subobject of the layout instance "*(the interface forms)*". Please refer to "Layouts" later in this section.

```
(define-class test-form-class
    :inherit-from(ui-form-class)
    :properties(
                label "Ui-form-class Instance"
                x-offset 300
                y-offset 200
                width 300
                height 300
                measurement 'percentage
                )
    :subobjects(
                (button :class 'ui-action-button-class
                        x-offset 40
                        y-offset 40
                        width 50
                        height 10
                        )
                ;; when not specified, the measurement property is derived
                ;; from the superior.
                )
    )
```

```
AML> (add-object (the interface forms) 'test-form 'test-form-class)
#<TEST-FORM-CLASS @ #x21c6b3fa>
AML> (display (the interface forms test-form))
T
```

Please refer to the class hierarchy tree of the *GUI base classes* reference manual for available component and form classes. The manual contains description of the purpose and behavior of all instantiable widgets as well as source code examples.

6.1.2 Layouts

A well-designed AML application should separate its GUI object tree from the model tree. An AML application developer should group its *top-level* forms in layouts. A layout of class *ui-layout-class* is typically instantiated as a subobject of the existing *"(the model-manager inteface)"* object. *"(The model manager interface)"* is defined with the purpose of being the parent of all GUI layouts.

Example

Given that *application-form1-class, application-form2-class…* are application form classes defined by the application developer, below is an example of defining and instantiating a layout.

```
(define-class missile-design-system-layout
    :inherit-from(ui-layout-class)
    :properties(
                ;; new developer defined properties
                screen-size (get-screen-size)
                viewport-x-offset 10
                viewport-y-offset 20
                viewport-width    (- (first ^screen-size) 20)
                viewport-height   (- (second ^ screen-size) 20)
                )
    :subobjects(
                (form1 :class 'application-form1-class
                    x-offset ^^viewport-x-offset
                    y-offset ^^viewport-y-offset
                    width 500
                    height 500
                    label "Form 1 version 1"
                    )
                (form2 :class 'application-form2-class
                    x-offset ^^viewport-x-offset
                    y-offset ^^viewport-y-offset
                    width 500
                    height 500
                    label "Form 2 version 1"
                    )
                …etc
                )
    )
(add-object (the interface) 'missile-design-layout 'missile-design-layout-class)
(display (the interface missile-design-layout form1))
…etc
```

Refer to the classes: *ui-layout-class*, *ui-form-class* and the methods *display*, *hide*, *raise* and *update* in the GUI base classes reference manual.

6.1.3 Box Model Example

The following example shows a box model managed by a form. Thanks to the *apply-action, cancel-action, ui-apply-button-class, ui-action-button-class* features, as well as the AML dependency mechanism, building a user interface to manage a model does not necessarily require the definition of specialized methods or functions.

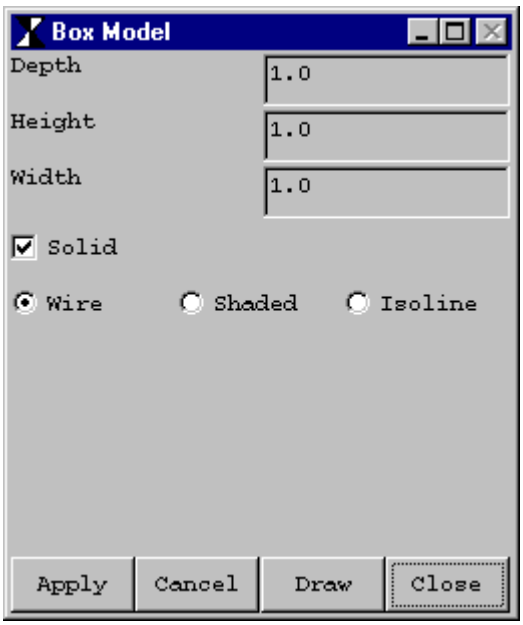
In the following example, the *box-model-form* requires the existence of a current model of class *box-model*.

The *apply* button applies the widget values of the form to the box model properties, i.e. no change is made to the box model unless the *apply* button is pressed. The *cancel* button resets the widget values of the form to the current property values of the box model, i.e. to the values that were last applied. The form can be a subobject of any *ui-layout-class* instance *(see ui-layout-class)*. For the sake of this example, it will be instantiated as a subobject of *"(the model-manager interface forms)"*.

```
(define-class box-model
     :inherit-from(box-object)
     )

(define-class box-model-form-class
     :inherit-from(ui-form-class)
     :properties(
     ;; Property created to keep a pointer to the box model
     ;; being modeled.
     ;; Note: The function root-object does not establish any
     ;; dependency, therefore the current-model property needs
     ;; to be smashed whenever the root object changes
     current-model (let* ((current-model (root-object))
                          )
                     (when (typep current-model 'box-model)
                     current-model))
     x-offset 50
     y-offset 50
     height 280
     width 250
     label "Box Model"
     measurement 'percentage ;; This is the default formula anyway
     )
     :subobjects(
          (bdepth :class 'ui-labeled-field-class
               x-offset 0 y-offset 0 width 100 height 10
               label "Depth"
               content (if ^^current-model (the depth (:from ^^current-model))
                           "N/A")
               apply-action (when ^^current-model
               '(change-value
               (the depth (:from ^^current-model))
               (get-value (the superior))))
               cancel-action '(smash-value ^content)
               )
          (bheight :class 'ui-labeled-field-class
                x-offset 0 y-offset 10 width 100 height 10
                label "Height"
                content (if ^^current-model
                            (the height (:from ^^current-model)) "N/A")
                apply-action (when ^^current-model
                                   '(change-value
                                     (the height (:from ^^current-model))
                                     (get-value (the superior))))
                                     cancel-action '(smash-value ^content)
                )
```

```
        (bwidth :class 'ui-labeled-field-class
                x-offset 0 y-offset 20 width 100 height 10
                label "Width"
                content (if ^^current-model
                    (the width (:from ^^current-model)) "N/A")
                apply-action (when ^^current-model
                                   '(change-value
                                       (the width (:from ^^current-model))
                                       (get-value (the superior))))
                cancel-action '(smash-value ^content)
                )
        (solid? :class 'ui-toggle-button-class
                x-offset 0 y-offset 30 width 100 height 10
                label "Solid"
                status (when ^^current-model
                         (the solid? (:from ^^current-model)))
                apply-action (when ^^current-model
                                    '(change-value
                                    (the solid? (:from ^^current-model))
                                    ^status))
                cancel-action '(smash-value ^status)
                )
    (render :class 'ui-radio-buttons-class
            x-offset 0 y-offset 40 width 100 height 10
            labels-list '("Wire" "Shaded" "Isoline")

            status (when ^^current-model
                     (case (the render (:from ^^current-model))
                          ('boundary 0)
                          ('shaded 1)
                          ('isoline 2)))
            apply-action (when (and ^^current-model ^status)
                          '(change-value
                              (the render (:from ^^current-model))
                              (nth ^status (list 'boundary 'shaded 'isoline))))
            cancel-action '(smash-value ^status)
            )
        (apply :class 'ui-apply-button-class
            x-offset 0 y-offset 90 width 25 height 10
            )
        (cancel :class 'ui-cancel-button-class
            x-offset 25 y-offset 90 width 25 height 10
            update-form? t)
        (draw :class 'ui-action-button-class
            x-offset 50 y-offset 90 width 25 height 10
            label "Draw"
            button1-action (when ^^current-model
                                 '(draw ^^current-model))
            button3-action (when ^^current-model
                                 '(undraw ^^current-model))
            )
        (close :class 'ui-action-button-class
            x-offset 75 y-offset 90 width 25 height 10
            label "Close"
            button1-action '(hide (the superior superior))
            )
        )
)

;; The following is a sample function to instantiate the form and display it or
;; to display it only if it has already been created. This function can be modified
;; by the developer to allow other parameters for the form properties

(defun display-box-model-form (box-model)
    (let* (
         (layout (the interface forms))
         (form (or
                (the box-model-form (:from layout :error nil))
                (add-object layout 'box-model-form 'box-model-form-class)
                ))
            )
```

```
    (when form
        (change-value (the current-model (:from form)) box-model)
        (display form))
    )
)

;; To create the model and display the form:
(create-model 'box-model)

(display-box-model-form (root-object))
;; The function root-object returns the current model which is the box-model
;; we just created.
```



6.1.4 Optional Exercise

As an exercise that combines both *ui-base-classes* and *ui-advanced-classes* techniques, define and instantiate the form in Figure 22 to interface the *missile-data-model-class* model. Figure 22 also shows the class of the different widgets on the form.

Hints:

1- The class *ui-model-tree*'s main input property is *root-object* that should point to the missile model instance.

2- The class *ui-model-tree's* main output property is *selected-item* that always hold the instance of the user selected object. (i.e. the object that the user selected by a mouse left-click)

3- The class *ui-multiple-property-subform-class* does not require the existence of a *data-model-node-mixin*. It only needs a list of current property objects specified in the property *property-objects-list*. Refer the *Model Interface Manual* for a description of the format of *property-objects-list* and an example.

4- Use the *ui-model-tree's button1-action* to assign the current *property-object-list* of the *ui-multiple-property-subform-class*.

5- The class *ui-apply-button-class* relies on the property *apply-form* to be linked to the form/subform it should apply.

6- The class *ui-cancel-button-class* relies on the property *cancel-form* to be linked to the form/subform it should reset.

7- If any modification occurs in the model, the method *update* should be called on the form in order to reflect the change.

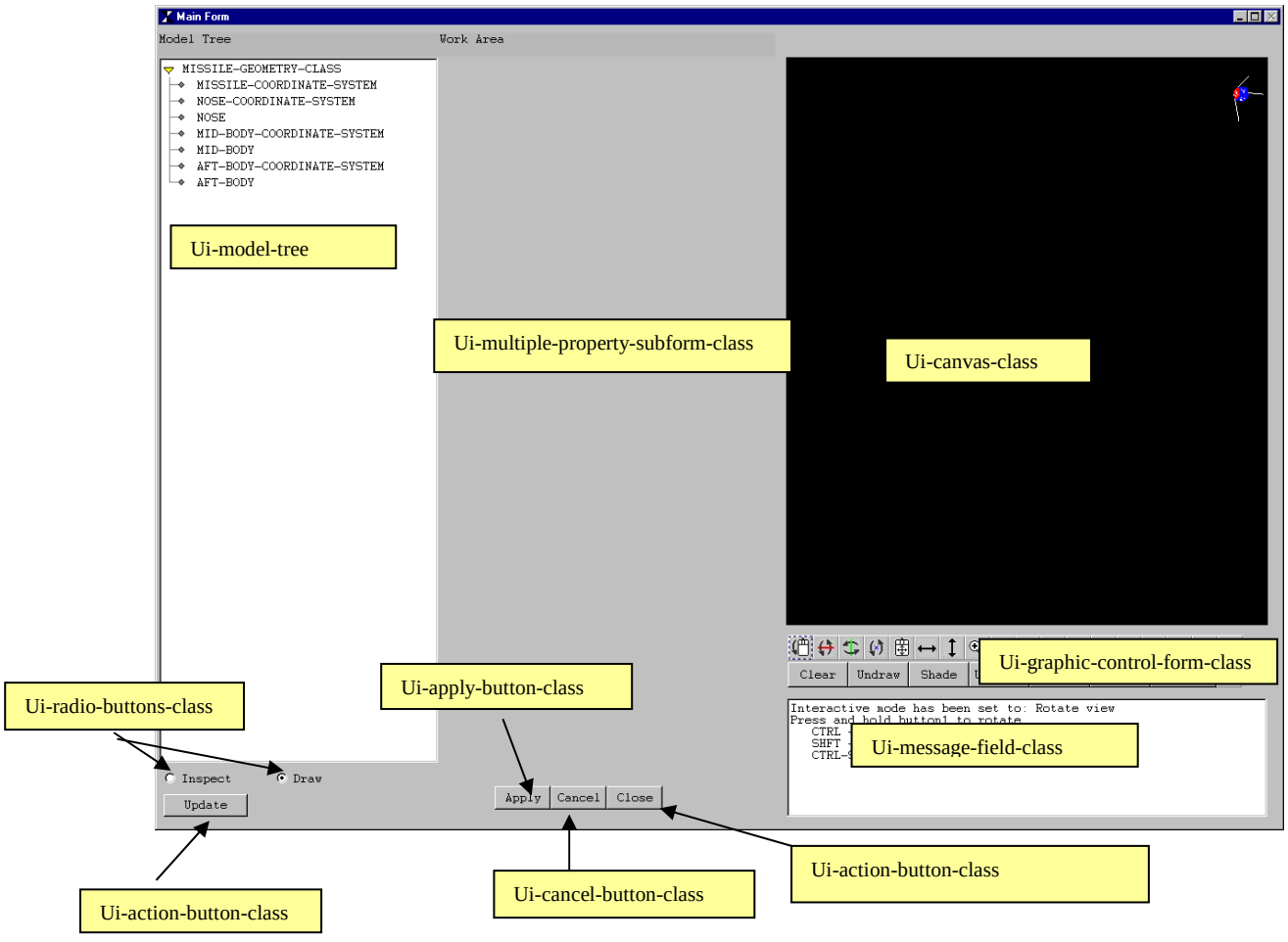
8- The *ui-graphic-control-form-class* should point to the instance of *ui-canvas-class* in order to activate it. Each canvas needs to be activated to become the *current-display*. The canvas can also be programmatically activated using the method *activate-display* after displaying the form. Refer to *ui-canvas-class* in the *GUI Base Classes* manual.

Note:

1- It is usually not efficient to create one instance of *ui-multiple-property-subform-class* with a *property-objects-list* that gets extensively modified at runtime because this leads to a lot of widget recreation and deletion. However, this will be accepted for the sake of this exercise.

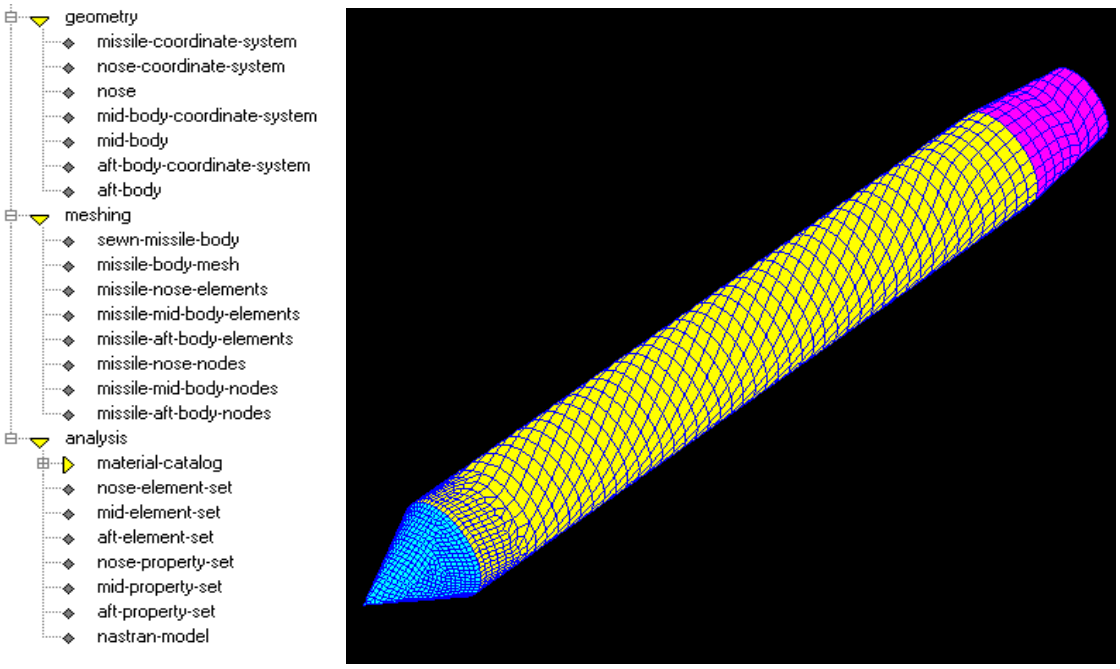
2- The functionality that this form provides the user with already exists (and more) with *ui-data-model-main-form*, so defining it here is only for the sake of exercise.

Figure 22: Form showing the class names of its constituents



7. Meshing and Analysis

Assuming the missile axis body component geometry has been created, an analysis portion of the model can be created. In this section, the missile geometry will be "tagged" and "meshed" to create input for a Nastran modal analysis. These concepts are explained in the following sections.



✓ The AML/Patran interface, AML/Analysis interface, and the AML/Nastran interface are not included in the standard AML release. They are additional AML modules that must be loaded separately. Therefore this functionality is not documented in the AML reference manual. These sections serve only as a portion of the systems' documentation.

`To perform the exercises in this section, you must have the` ***`analysis-module-pack-type-2 module loaded .`***

7.1 Attribute Tagging

The methodology of geometry attribute tagging and tag propagation allows resultant geometry from a boolean operation to refer back to the tags attached to the original geometry. In the context of meshing, attaching tag attributes to the geometry serves two purposes. The first purpose is to control mesh refinement on individual parts of the resultant geometry. The second is to provide the ability to query for mesh entities from the tagged geometry once the mesh has been generated.

New AML Constructs

- *tagging-object*
- *get-dimension-tags*

TAGGING-OBJECT

The tagging-object implements geometry attribute tagging and tag propagation through geometric operations. All geometric classes that inherit from this class will be tagged.

NOTE: The *tagging-object* must be the first element in the *:inherit-from (...)* list.

Properties:

id-tag — An identifier used to associate geometry with the object. This value is set by the system and should be only queried not set at this point. The value is set after the *geom* property has been demanded. **This value should not be manually overwritten or changed.**

tag-dimensions — This determines which entities of the geometry is to be tagged. The default *nil* means a solid object tags the solid, a surface tags the surfaces, a wire tags the edges, and points tags points. If the value is a list *'(0 1 2 3)*, all points, edges, surfaces, solids associated with a geometry will be tagged.

tag-attributes — Attribute list associated with the object. At present, these attributes are used in the context of meshing. **The list includes in order: maximum edge size, minimum edge size, curvature refinement value (0 for off 2 for on), curvature approximation error (in a percentage), segment value (0 for off 1 for on), segment size, and entity tolerance.** Default is *'(0.25 0.0625 0 0.1 0 10.0 1.0e-5)*. By default, these attributes will be applied to all tagged points, edges, faces, and solids associated with the geometry. **At the current release, only the maximum edge size attribute is used. The others are reserved.**

overwrite-other-tags? — This property determines whether the object will use its own tags or if it will use tags passed on to it. **This is only relevant for objects that get tags passed to them, for example, boolean**

objects, sweep objects, geom-copy objects, sub-geom objects, etc. Hence if this property is set to *t* for a *geom-copy-object*, its geom will have its own tags, and if it is *nil*, its geom will have the tags of the source object. Note that even if it is set to *t*, only the tags on sub-geoms of dimensions included in the *tag-dimensions* property will be overwritten.

GET-DIMENSION-TAGS [Method]

This method returns a list of tag attributes for all entities tagged on the geometry with a certain geometric dimension (1D, 2D, or 3D).

Format:

(get-dimension-tags object dimension)

Arguments:

object	Instance of type *tagging-object*.
dimension	Integer value of 1, 2, or 3.

Example:

```
(define-class tag-example-class
  :inherit-from (object)
  :properties (
               min-box-elem-size    0.06
               min-cyl-elem-size    0.03

               box-tags         (get-dimension-tags ^box 1)
               cylinder-tags    (get-dimension-tags ^cylinder 1)
               union-tags       (get-dimension-tags ^box-cyl-union 1)
               )
  :subobjects (
               (box :class '(tagging-object box-object)
                    tag-dimensions '(1 2)
                    tag-attributes (list ^^min-box-elem-size .1
                                         0 0.1 0 20.0 1.0e-5)
                    )
               (cylinder :class '(tagging-object cylinder-object)
                         diameter 0.3
                         tag-dimensions '(1 2)
                         tag-attributes (list ^^min-cyl-elem-size .1
                                              0 0.1 0 20.0 1.0e-5)
                         orientation    (list (translate '(0 0 1.5)))
                         )
               (box-cyl-union :class '(tagging-object union-object)
                              object-list    (list ^box ^cylinder)
                              tag-dimensions '(1 2)
                              overwrite-other-tags? nil
                              )
               )
  )
```

```
> (create-model 'tag-example-class)
#<TAG-EXAMPLE-CLASS @ #x233375fa>
> (the box-tags)
((20073039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20074039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20075039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20076039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20077039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20078039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20079039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20080039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20081039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20082039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20083039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20084039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0))
> (the cylinder-tags)
((20085040 0.03 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20086040 0.03 0.1 0 0.1 0 20.0 10.0e-6 0))
> (the union-tags)
 ((20086040 0.03 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20073039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20074039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20075039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20076039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20077039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20078039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20079039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20080039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20081039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20082039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20083039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20084039 0.06 0.1 0 0.1 0 20.0 10.0e-6 0)
 (20085040 0.03 0.1 0 0.1 0 20.0 10.0e-6 0))
> (change-value (the box-cyl-union overwrite-other-tags?) t)
T
> (the union-tags)
((20087041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0)
 (20088041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0)
 (20089041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0)
 (20090041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0)
 (20091041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0)
 (20092041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0)
 (20093041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0)
 (20094041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0)
 (20095041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0)
 (20096041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0)
 (20097041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0)
 (20098041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0)
 (20099041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0)
 (20100041 0.25 0.0625 0 0.1 0 10.0 10.0e-6 0))
```

The *tag-union-example* object illustrates tag propagation for the union of a box and a cylinder and the use of element size refinement to refine the tags on the edges of the cylinder. Since the *overwrite-other-tags?* property on the union is set to *nil* in the beginning, any edge tags passed from the box or the cylinder to the union will not be overwritten by the tags set on the union. Notice that once the *overwrite-other-tags?* property was changed to *t*, the union object's tags were used by the defaults given from the initial class definition of *tagging-object*. It is important to note that this example must have the *simplify?* property in the *box-cyl-union* set to *t* because the common face between the *box* and *cylinder* needs to be included in the model.

7.2 Meshing

Meshing of a geometric model in AML is achieved through a single class, the *patran-mesh-interface-class*. The geometry to be meshed is given to this object and it in turn creates the necessary information to pass to the meshing process.

New AML Constructs

- patran-mesh-interface-class
- meshdb-class
- *load-mesh*
- *draw-mesh*
- logical-path entries for the :*patran-meshdb-interface* meshing system

---

PAVER-MESH-CLASS **[CLASS]**

This object is used to generate a mesh for a specified geometric object. After the mesh is generated, the query objects described below can be used to retrieve nodes, edges, faces, and regions from the mesh.

Properties:

object-to-mesh	The geometric object which is to be meshed. Default is *nil*.
logical-path	An entry in the logical path file which points to the path to which all mesh files are written. Default is *:meshes*.
mesh	When demanded, this property generates the mesh and returns *t* when the mesh operation was successful.
solid-mesh?	Determines if the mesh is a solid mesh or a surface mesh. Default is *t* for a solid mesh.

MESHDB-CLASS **[CLASS]**

An instance of this class is a mesh database used to store mesh data generated by an instance of the patran-mesh-interface-class.

Properties:

db-id	An ID of the mesh database. This should not be set or changed by the user.
db-name	Name of the mesh database (string).

LOAD-MESH **[Method]**

This method retrieves the mesh data from the mesh files and adds objects under the *paver-mesh-class object* which are used to display the mesh boundaries and edges.

Format:

(load-mesh object)

Arguments:

object	An instance of type *paver-mesh-class*.

DRAW-MESH **[Method]**

This method is used to visualize the mesh boundaries and edges.

Format:

(draw-mesh mesh-object [:visible? t] [:boundary? nil] [:shade? nil] [:update? nil])

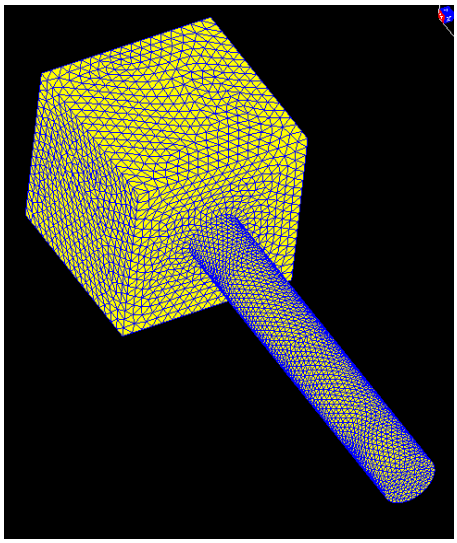
Arguments:

visible?	Keyword which determines whether the drawn mesh is visible or not. Default is *t*.
edge?	Keyword which determines if the mesh edges are drawn. Default is *nil*.
boundary?	Keyword which determines if the faces on the boundary of the mesh are drawn. Default is *nil*.
shade?	Keyword which determines if the boundary faces of the mesh are shaded. Default is *nil*.
update?	This keyword determines whether the graphics window is updated once the mesh has been drawn. Default is *t*.

Example:

```
(define-class mesh-example-class
  :inherit-from (object)
  :properties (
           min-box-elem-size 0.06
           min-cyl-elem-size 0.03
           box-tags (get-dimension-tags ^box 1)
           cylinder-tags (get-dimension-tags ^cylinder 1)
           union-tags (get-dimension-tags ^box-cyl-union 1)
           )
```

```
  :subobjects (
           (box :class '(tagging-object box-object)
            tag-dimensions '(1 2)
            tag-attributes (list
                    ^^min-box-elem-size
                    .1
                    0
                    0.1
                    0
                    20.0
                    1.0e-5
                    )
            )
           (cylinder :class '(tagging-object cylinder-object)
             diameter 0.3
             tag-dimensions '(1 2)
             tag-attributes (list  ^^min-cyl-elem-size
                                .1
                                0
                                0.1
                                0
                                20.0
                                1.0e-5)
             orientation (list
                      (translate '(0 0 1.5))
                      )
             )
           (split-cylinder-sheet :class 'sheet-object
                     width (* 2 (the superior superior cylinder height))
                     height (* 2 (the superior superior cylinder diameter))
                     orientation (list
                          (rotate 90 :y)
                          (translate (list
                            0
                            0
                            (half ^width)
                                  )
                                 )
                          )
                     )
           (imprinted-cylinder :class 'geometry-with-split-periodic-faces-
class
                   source-object ^^cylinder

                                 )
           (box-cyl-union :class '(tagging-object union-object)
                  object-list (list
                       ^^box
                       ^^imprinted-cylinder
                       )
                  tag-dimensions '(1 2)
                  overwrite-other-tags? nil
                  simplify? nil
                  )
           (mesh-database :class 'meshdb-class
                  )
           (analysis-mesh :class 'paver-mesh-class
                  mesh-database-object ^^mesh-database
                  object-to-mesh ^^box-cyl-union
                  solid-mesh? nil
                  )
           )
  )
AML> (create-model 'mesh-example-class)
#<MESH-EXAMPLE-CLASS @ #x22f43032>
```



7.3 Mesh Queries

Meshes can be filtered so that only certain portions of them are shown, along with their options for being display. The different query objects facilitate this feature of AML.

New AML Constructs

- *mesh-query-class*
- *write-data-file*
- *mesh-nodes-query-class*
- *mesh-elements-2d-query-class*

MESH-QUERY-CLASS **[Class]**

All query objects inherit from *mesh-query-class*. This class should not be instantiated. It is only documented to show the origination place of the common properties within the mesh queries.

Properties:

mesh-object	Object of type *paver-mesh-class* containing mesh information. Default is *nil*.
mesh-database-object	Object of type *meshdb-class* containing the stored mesh information. Default is *nil*.
tagged-object-list	List of tagged objects from which mesh entities are to be retrieved.
tag-dimensions-list	Determines the type of mesh entity to be retrieved from the objects specified in the *tagged-object-list* property. 0 for nodes, 1 for edges, 2 for faces, and 3 for solids.

MESH-NODES-QUERY-CLASS **[Class]**

This object can be used to retrieve nodes from the objects specified in the *tagged-object-list* property.

Inherit-From:

mesh-query-class

Properties:

num-nodes	Number of nodes in the mesh query.
mesh-entities-list	A list of the node ids in the mesh query.

MESH-ELEMENTS-2D-QUERY-CLASS **[Class]**

This object can be used to retrieve mesh faces from the objects specified in the *tagged-object-list* property.

Inherit-From:

mesh-query-object

Properties:

num-elements	Number of elements in the mesh query.
mesh-entities-list	A list of the element ids in the mesh query.
type-list	Set to '(:triangle :quad), specifying that triangular and quadrilateral elements may exist in the surface query.
tag-dimensions-list	Set to ‘(2) specifying that geometry entities of dimension 2 are corresponding to the surface mesh query.

Example:

```
(define-class  mesh-queries-example-class

  :inherit-from (object)

  :properties (

               min-box-elem-size 0.06

               min-cyl-elem-size 0.03

               box-tags (get-dimension-tags ^box 1)

               cylinder-tags (get-dimension-tags ^cylinder 1)

               union-tags (get-dimension-tags ^box-cyl-union 1)

               )

  :subobjects (

               (box :class '(tagging-object box-object)

                   tag-dimensions '(1 2)

                   tag-attributes (list ^^min-box-elem-size .1

                                        0 0.1 0 20.0 1.0e-5)

                   )

               (cylinder :class '(tagging-object cylinder-object)

                   diameter 0.3

                   tag-dimensions '(1 2)
```

```
                tag-attributes (list ^^min-cyl-elem-size .1
                                     0 0.1 0 20.0 1.0e-5)
                orientation (list (translate '(0 0 1.5)))
                )
             (split-cylinder-sheet :class 'sheet-object
                                   width (* 2 (the superior superior cylinder
height))
                                   height (* 2 (the superior superior cylinder
diameter))
                                   orientation (list
                                               (rotate 90 :y)
                                               (translate (list
                                                          0
                                                          0
                                                          (half ^width)
                                                          )
                                                         )
                                               )
                                   )
            (imprinted-cylinder :class 'geometry-with-split-periodic-faces-
class
                                source-object ^^cylinder

                                                )
             (box-cyl-union :class '(tagging-object union-object)
                tag-dimensions '(1 2)
                overwrite-other-tags? nil
                simplify? nil
                )
             (mesh-database :class 'meshdb-class
                )
```

```
        (analysis-mesh :class 'paver-mesh-class
            mesh-database-object ^^mesh-database
            object-to-mesh ^^box-cyl-union
            solid-mesh? nil
            )
        (box-elements :class 'mesh-elements-2d-query-class
            tagged-object-list (list ^^box)
            mesh-object ^^analysis-mesh
            color 'magenta
            render 'boundary-shaded


            )
        (cylinder-nodes :class 'mesh-nodes-query-class
            tagged-object-list (list ^^cylinder)
            mesh-object ^^analysis-mesh
            color 'green
            )
        )
)
```

Code Explanation

The *box-elements* and *cylinder-nodes* objects allow the user to view the respective portions of the mesh from the *analysis-mesh* object. These can be used to specify various regions of a part representing different materials or specific properties of a part.

Exercise 3a

After a close review of the class definitions and AML constructs described in the previous section, create a meshing model for the missile geometry used in exercise 3.

Two underlying classes need to be defined with tagging to ensure the appropriate tagging properties will be propagated in the model; they are *'(tagging-object open-truncated-cone-class)* and *'(tagging-object open-cone-split-class)*. Create the *'(tagging-object open-truncated-cone-class)* in order to provide a "tagged" counterpart to the *open-truncated-cone-object*. Create the *'(tagging-object open-cone-split-class)* such that it will embed an edge in the nose object (essentially dividing it into two symmetric topological pieces) to ensure the meshing program can automatically pave a mesh on the cone. Modify the *missile-geometry-class* to have three properties to control each body component's maximum element size as given in the *tag-attributes* property. Create a *missile-mesh-model-class* to encapsulate a geometric entity to mesh, a surface *:hex* mesh, along with element and node queries for each of the missile body components.

Use the following Instance diagram as a guide to build the necessary classes.

Instance Diagrams for Exercise 3a

```
tagged-open-truncated-cone-class [tagging-object open-truncated-cone-object]
   missile-general-body-radius
   missile-nose-length
   missile-nose-radius
   missile-mid-body-length

tagged-open-cone-split-class [tagging-object imprint-class]
   height
   diameter
   target-object
   tool-object-list
   splitting-plane [sheet-object]
   cone [open-cone-object]

missile-geometry-class [object]
   missile-general-body-radius
   missile-nose-length
   missile-nose-radius
   missile-mid-body-length
   missile-mid-body-radius
   missile-aft-body-length
   missile-aft-body-start-radius
   missile-aft-body-end-radius
   missile-nose-max-element-size
   missile-mid-body-max-element-size
   missile-aft-body-max-element-size
   missile-coordinate-system [coordinate-system-class]
   nose-coordinate-system [coordinate-system-class]
        origin
        reference-coordinate-system
   nose [tagged-open-cone-split-class]
        diameter
        height
        orientation
        reference-coordinate-system
        max-element-size
        tag-dimensions
        tag-attributes
```

```
   mid-body-coordinate-system [coordinate-system-class]
        origin
        reference-coordinate-system
   mid-body [tagged-open-cylinder-object]
        diameter
        height
        orientation
        reference-coordinate-system
        max-element-size
        tag-dimensions
        tag-attributes
   aft-body-coordinate-system [coordinate-system-class]
        origin
        reference-coordinate-system
   aft-body [tagged-open-truncated-cone-object]
        start-diameter
        end-diameter
        height
        orientation
        reference-coordinate-system
        max-element-size
        tag-dimensions
        tag-attributes

missile-mesh-model-class
   nose-object
   mid-body-object
   aft-body-object
   sewn-missile-body [sewn-object]
        object-list
   missile-body-mesh [patran-mesh-object]
        object-to-mesh
        element-shape
        solid-mesh?
   missile-nose-elements [patran-2d-mesh-entities-query-object]
        tagged-object-list
        mesh-object
   missile-mid-body-elements [patran-2d-mesh-entities-query-object]
        tagged-object-list
        mesh-object
   missile-aft-body-elements [patran-2d-mesh-entities-query-object]
        tagged-object-list
        mesh-object
   missile-nose-nodes [patran-0d-mesh-entities-query-object]
        tagged-object-list
        mesh-object
   missile-mid-body-nodes [patran-0d-mesh-entities-query-object]
        tagged-object-list
        mesh-object
   missile-aft-body-nodes [patran-0d-mesh-entities-query-object]
        tagged-object-list
        mesh-object

missile-model-class
   geometry [missile-geometry-class]
   meshing [missile-mesh-model-class]
        nose-object
        mid-body-object
        aft-body-object
```

Exercise 3a Solution

```
(in-package :aml)

(define-class tagged-open-truncated-cone-class
  :inherit-from (tagging-object open-truncated-cone-object)
  )

(define-class tagged-open-cone-split-class
  :inherit-from (tagging-object imprint-class)
  :properties (
           height    1.0
           diameter  1.0

           target-object    ^cone
           tool-object-list (list ^splitting-plane)
           (splitting-plane :class 'sheet-object
            height ^^diameter
            width  ^^height
            orientation (list (rotate 90 '(0 1 0)))
            )
           (cone :class 'open-cone-object
             height ^^height
             diamter ^^diameter
             )
           )
  )

(define-class missile-geometry-class
  :inherit-from (object)
  :properties (
               missile-general-body-radius      1.0

               missile-nose-length              2.0
               missile-nose-radius              ^missile-general-body-radius

               missile-mid-body-length          14.0
               missile-mid-body-radius          ^missile-general-body-radius

               missile-aft-body-length          2.0
               missile-aft-body-start-radius    ^missile-general-body-radius
               missile-aft-body-end-radius      0.75

               missile-nose-max-element-size       0.1
               missile-mid-body-max-element-size   0.25
               missile-aft-body-max-element-size   0.25

               )
  :subobjects (
               (missile-coordinate-system :class 'coordinate-system-class
                origin (list 0.0 0.0 0.0)
                )

               (nose-coordinate-system :class 'coordinate-system-class
                 origin    (list (* 0.5 ^^missile-nose-length) 0.0 0.0)
                 reference-coordinate-system ^^missile-coordinate-system
                )

               (nose :class 'tagged-open-cone-split-class
                 height              ^^missile-nose-length
                 diameter            (* ^^missile-nose-radius 2.0)
                 orientation         (list
                                      (rotate -90 '(0 1 0))
                                      )
                 reference-coordinate-system ^^nose-coordinate-system
```

```
      max-element-size            ^^missile-nose-max-element-size
      tag-dimensions   '(1 2)
      tag-attributes   (list ^max-element-size 0.0625 0 0.1 0 10.0 1.0e-5 0)
                 )

               (mid-body-coordinate-system :class 'coordinate-system-class
                origin          (list
                                    (+ (/ ^^missile-nose-length 2.0)
                                       (/ ^^missile-mid-body-length 2.0)
                                        )
                                    0.0
                                    0.0)
                reference-coordinate-system ^^nose-coordinate-system
                )

               (mid-body :class 'tagged-open-cylinder-object
                 diameter                (* ^^missile-mid-body-radius 2.0)
                 height                     ^^missile-mid-body-length
                 orientation                (list
                                             (rotate 90 '(0 1 0))
                                             )
                 reference-coordinate-system ^^mid-body-coordinate-system
      max-element-size            ^^missile-mid-body-max-element-size
      tag-dimensions   '(1 2)
      tag-attributes   (list ^max-element-size 0.0625 0 0.1 0 10.0 1.0e-5 0)
                 )

               (aft-body-coordinate-system :class 'coordinate-system-class
                 origin                     (list
                                             (+ (/ ^^missile-mid-body-
length 2.0)
                                                (/ ^^missile-aft-body-
length 2.0)
                                                )
                                             0.0
                                             0.0)
                 reference-coordinate-system ^^mid-body-coordinate-system
                )

               (aft-body :class 'tagged-open-truncated-cone-class
                 start-diameter     (* ^^missile-aft-body-start-radius 2.0)
                 end-diameter       (* ^^missile-aft-body-end-radius 2.0)
                 height                     ^^missile-aft-body-length
                 orientation                (list
                                             (rotate 90 '(0 1 0))
                                             )
                 reference-coordinate-system ^^aft-body-coordinate-system
      max-element-size            ^^missile-aft-body-max-element-size
      tag-dimensions   '(1 2)
      tag-attributes   (list ^max-element-size 0.0625 0 0.1 0 10.0 1.0e-5 0)
                 )

               )
  )

(define-class missile-mesh-model-class
  :inherit-from (object)
  :properties (
               nose-object        nil
               mid-body-object    nil
               aft-body-object    nil
                (node-set :class 'analysis-node-set-class
                    query-objects-list (list (the nodes-query (:from
^^missile-body-mesh)))
                 )
           )
```

```
  :subobjects (
               (sewn-missile-body :class 'sewn-object
                object-list (list
                             ^^nose-object
                             ^^mid-body-object
                             ^^aft-body-object
                             )
                )
 (imprint-sewn-missile-body :class 'geometry-with-split-periodic-faces-class
                    source-object  ^^sewn-missile-body
                    )
               (mesh-database :class 'meshdb-class
                     )
       (missile-body-mesh :class 'paver-mesh-class
                object-to-mesh    ^^sewn-missile-body
                mesh-database-object ^^mesh-database
                element-shape     :hex
                solid-mesh?       nil
                )

       (missile-nose-elements :class 'mesh-elements-2d-query-class
                tagged-object-list (list ^^nose-object)
                mesh-object        ^^missile-body-mesh
                color 'cyan
                render 'boundary-shaded
                )

     (missile-mid-body-elements :class 'mesh-elements-2d-query-class
                tagged-object-list (list ^^mid-body-object)
                mesh-object        ^^missile-body-mesh
                color 'yellow
                render 'boundary-shaded
                )

     (missile-aft-body-elements :class 'mesh-elements-2d-query-class
                tagged-object-list (list ^^aft-body-object)
                mesh-object        ^^missile-body-mesh
                color 'magenta
                render 'boundary-shaded
                )

     (missile-nose-nodes :class 'mesh-nodes-query-class
                tagged-object-list (list ^^nose-object)
                mesh-object        ^^missile-body-mesh
                color 'green
                )

     (missile-mid-body-nodes :class 'mesh-nodes-query-class
                tagged-object-list (list ^^mid-body-object)
                mesh-object        ^^missile-body-mesh
                color 'magenta
                )

     (missile-aft-body-nodes :class 'mesh-nodes-query-class
                tagged-object-list (list ^^aft-body-object)
                mesh-object        ^^missile-body-mesh
                color 'lightblue
                )
               )
  )

(define-class missile-model-class
  :inherit-from (object)
```

```
:properties (

          )
:subobjects (
          (geometry :class 'missile-geometry-class

            )

          (meshing :class 'missile-mesh-model-class
           nose-object        (the nose (:from ^^geometry))
           mid-body-object    (the mid-body (:from ^^geometry))
           aft-body-object    (the aft-body (:from ^^geometry))
           )

          )
)
```

Code Explanation

Two underlying classes were defined to ensure the appropriate tagging properties are propagated in the model. They are controlled through the *max-element-size* property on each instance of a *tagging-object* which uses a *the*-reference to obtain a value given in the *missile-geometry-class*. In the *tagged-open-truncated cone-class*, notice that *tagging-object* is specified first in the inheritance list. This ensures the proper attribute tagging and propagation through the model as Boolean operations are performed on the geometry. The *tagged-open-cone-split-class* needs to be defined to embed an edge in the nose object to ensure the meshing program can automatically pave a mesh on the cone. The meshing application fails without this embedded edge, thus demonstrating the focus on preparing various representations of geometry and topology for the meshing and analysis models. The *tagged-open-cone-split-class* uses an *imprint-class* to place an edge in the parametric domain space of the cone. Internally, the meshing application needs to have this in order to create a successful "paved" mesh.

The *missile-mesh-model* has three "pointer" properties that will be overwritten on instantiation to point to the *nose, mid-body, or aft-body* respectively. This technique enables the ability to make this class more modular and robust. The *sewn-missile-body* creates one piece of geometry that contains all of the individually tagged entities to be passed to the meshing application. The mesh is interfaced through the *missile-body-mesh* object which specifies a "hex" surface mesh. This object manages all communication with the meshing application and organizes data for the individual queries. The 0D and 2D queries respectively interrogate the mesh for the nodes and elements corresponding within each tagged geometric object.

7.4 Finite Element Analysis

The following AML code demonstrates a typical analysis model using the AML Nastran Interface. It shows the typical classes used in an AML analysis such as interfaces with NASTRAN, ANSYS, and LSDYNA. The analysis interfaces are based on a system called *:analysis-interface* which is the core virtual layer to the various applications. The model is created using classes/methods/functions from this system and classes exist within the various application interfaces (e.g. *:nastran-interface*) that can interrogate the base analysis classes/methods/functions for specific implementation.

New AML Constructs

- *analysis-model-class*
- *material-catalog-class*
- *analysis-property-set-2d-type-1-class*
- *analysis-element-set-2d-type-1-class*
- *analysis load classes*
- *analysis constraint classes*
- *analysis-load-case-class*
- *nastran-analysis-class*
- *analysis-post-processing-structural-linear-static-nastran-class*

ANALYSIS-MODEL-CLASS **[Class]**

This is the base class which manages communication and all interfaces with the analysis application. All node sets, element sets, property sets, load cases (boundary conditions), analysis types, and materials are specified in this class. When instantiated, application specific analysis classes query the properties of this class to determine their respective interfaces to the AML model.

Various analyses are available in the different application implementations. They are described below with their corresponding AML keyword used in the *analysis-type* property of the *analysis-model-class*, a description, and their corresponding analysis type in Nastran and Ansys:

AML Keyword	Description	Nastran "SOL"	Ansys "ANTYPE"
:modal	Normal Modes Analysis (Frequency Response)	103	2
:buckling	Static Buckling Analysis	105	1
:linear-static	Linear Static Structural Analysis (Stress/Deflection)	101	0
:static-aeroelastic-response	Static Aeroelastic Response (Aero/Structural Coupling)	144	N/A

Properties

analysis-type — Specifies the type of analysis to be performed. This is listed in the AML Keyword column of the table above.

mesh-object	A reference to an instance of a *patran-mesh-interface-class* (default nil)
material-catalog-object	A reference to an instance of an *material-catalog-class* (default nil)
load-case-objects-list	A list of instances of load case objects which inherit from *analysis-load-case-class* (default nil)
materials-list	A list specifying the materials used within the instances of the *analysis-property-set-class* (default nil)
property-set-objects-list	A list of instances of property set objects which inherit from *analysis-property-set-class* (default nil)
element-set-3d-objects-list	A list of instances of solid element set objects which inherit from *analysis-element-set-3d-class* (default nil)
element-set-2d-objects-list	A list of instances of surface element set objects which inherit from *analysis-element-set-2d-class* (default nil)
element-set-1d-objects-list	A list of instances of bar/beam/rod element set objects which inherit from *analysis-element-set-1d-class* (default nil)
node-set-objects-list	A list of instances of class *analysis-node-set-class*.

MATERIAL-CATALOG-CLASS [Class]

The AML analysis interface supports the use of a simple material catalog data file. The file is written in XML format with tags for the attributes of the material. Following the format of the sample data file below, the file can be customized to include any number of materials. The *:analysis-interface* contains a sample material file named “materials.xml” and is referenced by (system-resource :material-catalog "data" "materials.xml"). A portion of it is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="tsiMaterialCatalog.xsl"?>
<tsiMaterialCatalog                              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="tsiMaterialCatalog.xsd">
   <materialCatalogVersion>1.0</materialCatalogVersion>
   <materialCatalogComments>Standard materials catalog</materialCatalogComments>
   <materials>
      <material name="steel" type="steel">
         <materialType>linear-elastic</materialType>
         <materialClass>isotropic</materialClass>
         <materialComments></materialComments>
         <materialProperties>
            <materialProperty   name="elastic-modulus"   value="2.973332E7"   units="(lb   (in   -2))"
description="youngs modulus"></materialProperty>
            <materialProperty          name="poissons-ratio"          value="0.29"          units="nil"
description=""></materialProperty>
            <materialProperty     name="mass-density"     value="0.28"     units="(lb     (in     -3))"
description=""></materialProperty>
            <materialProperty    name="shear-modulus"    value="1.15245E7"    units="(lb    (in    -2))"
description=""></materialProperty>
         </materialProperties>
      </material>
      <material name="aluminum" type="aluminum">
         <materialType>linear-elastic</materialType>
         <materialClass>isotropic</materialClass>
         <materialComments></materialComments>
         <materialProperties>
            <materialProperty    name="elastic-modulus"    value="1.06E7"    units="(lb    (in    -2))"
description="youngs modulus"></materialProperty>
```

```
            <materialProperty          name="poissons-ratio"          value="0.33"          units="nil"
description=""></materialProperty>
            <materialProperty     name="mass-density"     value="0.098"     units="(lb     (in     -3))"
description=""></materialProperty>
            <materialProperty     name="shear-modulus"     value="4.0E6"     units="(lb     (in     -2))"
description=""></materialProperty>
         </materialProperties>
      </material>
      <material name="t300/5208" type="graphite/epoxy">
         <materialType>linear-elastic</materialType>
         <materialClass>orthotropic</materialClass>
         <materialComments></materialComments>
         <materialProperties>
            <materialProperty  name="elastic-modulus-longitudinal"  value="2.625E7"  units="(lbf  (in  -
2))" description=""></materialProperty>
            <materialProperty  name="elastic-modulus-lateral"  value="1.494E6"  units="(lbf  (in  -2))"
description=""></materialProperty>
            <materialProperty    name="shear-modulus"    value="1.04E6"    units="(lbf    (in    -2))"
description=""></materialProperty>
            <materialProperty          name="poissons-ratio"          value="0.28"          units="nil"
description=""></materialProperty>
            <materialProperty     name="mass-density"     value="1.6"     units="(g     (cm     -3))"
description=""></materialProperty>
            <materialProperty          name="ply-thickness"          value="0.005"          units="(in)"
description=""></materialProperty>
         </materialProperties>
      </material>
  </materials>
</tsiMaterialCatalog>
```

Properties

material-catalog-file-name	A string containing the full path to the material file (default (system-resource :material-catalog "data" "materials.xml"))

ANALYSIS-NODE-SET-CLASS [Class]

The *analysis-node-set-class* contains information a list of nodes queries. This class is used to specify which nodes will eventually be used in an analysis by setting the node-set-objects-list property on an instance of the *analysis-model-class*. Typically there will be one instance of *analysis-node-set-class* per instance of *patran-mesh-interface-class*, and the query-objects-list property of the *analysis-node-set-class* will simply contain a list of the single nodes-query subobject of the *patran-mesh-interface-class* instance.

Properties

query-objects-list	A list of instances of *mesh-nodes-query*-class.

ANALYSIS-PROPERTY-SET-2D-TYPE-1-CLASS [Class]

The *analysis-property-set-class* contains information about and specifies certain material properties of a geometry representation of a part and is inherited (mixed) into all property set classes. These properties depend on the type of geometry being defined. For example, sheet metal could be represented using the *analysis-property-set-2d-type-1-class*. Each usage comes with a set of specific assumptions. In the case of the *analysis-property-set-2d-type-1-class*, it is assumed that the material is isotropic with a constant thickness. Other types of property sets are available which have other material properties and assumptions (differentiated by the “type” specified in the class name).

Inherit From

analysis-property-set-class

Properties

material-catalog-object	A reference to an instance of *material-catalog-class*. This property comes from the *analysis-property-set-class* (default nil)
material-name	The name of the material given in the material catalog (default nil)
thickness	A number specifying the thickness of the material. The thickness is assumed constant. (default 1.0)

ANALYSIS-ELEMENT-SET-2D-TYPE-1-CLASS [Class]

The *analysis-element-set-class* contains information about and specifies the element type representing a portion geometry on a part and is inherited (mixed) into all element set classes. These elements depend on the type of geometry being specified. For example, sheet metal could be represented using the *analysis-element-set-2d-type-1-class*. These are also known as 2D, "shell", or "plate" elements. Each usage comes with a set of specific assumptions. In the case of the *analysis-element-set-2d-type-1-class*, it is assumed that the element is either three or four sided (tri or quad) in shape consisting of three or four nodes. The shape and number of nodes are dictated by the elements derived from the corresponding mesh query. The material property behavior of the elements is specified through the link to an instance of a *analysis-property-set-class.* Other types of element sets are available which have other assumptions (differentiated by the "type" specified in the class name).

Inherit From

analysis-element-set-class

Properties

property-set-object	A reference to an instance of an *analysis-property-set-class*. This property comes from the *analysis-element-set-class* (default nil)
query-objects-list	A list of references to instances of a *mesh-query-class*. This property comes from the *analysis-element-set-class*. The elements specified in these queries objects will be represented in the analysis with the material properties specified in the *property-set-object*. (default nil)

ANALYSIS LOAD CLASSES [Class]

A load in a finite element model could be force, moment, pressure, heat, magnetic flux, etc. This class serves as a general class that is inherited into all other load classes. Several loading classes

are available to represent various physical loading situations such as: *analysis-load-force-nodal-class, analysis-load-force-distributed-uniformly-nodal-class, analysis-load-moment-nodal-class, analysis-load-force-distributed-class, analysis-load-moment-distributed-class, analysis-load-force-distributed-1d-class,* and *analysis-load-pressure-2d-3d-class*.

Properties

target-object	A reference to an instance of a *mesh-query-class*. This represents the region of the geometry where the load is applied. (default nil)
load-vector	A list of x, y, and z components of the load vector. For example → *(list 10.45 0.86 34.5)* (default nil)

ANALYSIS CONSTRAINT CLASSES [Class]

A constraint in a finite element model constrains the model in some way such as restricting motion in translation or rotation, or constraining two points to be coincident throughout the simulation. This class serves as a general class that is inherited into all other constraint classes. Two constraint classes are available to represent various physical constraint situations such as: *analysis-constraint-displacement-class* and *analysis-constraint-displacement-type-2-class*.

Properties

target-object	A reference to an instance of a *mesh-query-object*. This represents the region of the geometry where the constraint is applied. (default nil)
tx	A number specifying the amount of translational displacement allowed in the x-direction. (default nil)
ty	A number specifying the amount of translational displacement allowed in the y-direction. (default nil)
tz	A number specifying the amount of translational displacement allowed in the z-direction. (default nil)
mx	A number specifying the amount of rotational displacement allowed in the x-direction. (default nil)
my	A number specifying the amount of rotational displacement allowed in the y-direction. (default nil)
mz	A number specifying the amount of rotational displacement allowed in the z-direction. (default nil)

ANALYSIS-LOAD-CASE-CLASS [Class]

In finite element analyses, a part is often loaded and constrained in several combinations of ways. Each combination represents a load case which consists of a set of loads and a set of constraints. This class manages these boundary conditions.

Inherit From

analysis-element-set-class

Properties

constraint-objects-list	A list of references to instances of an *analysis-constraint-class*. (default nil)
load-objects-list	A list of references to instances of an *analysis-load-class*. (default nil)

ANALYSIS-POST-PROCESSING-STRUCTURAL-LINEAR-STATIC-NASTRAN-CLASS [Class]

This class produces post processing plots from a linear-static analysis in Nastran.

Properties

mesh-database-object	Refers to an instance of *meshdb-class*.
mesh-query-objects-list	A list of mesh queries on which post processing results will be displayed.
analysis-interface-nastran-object	Refers to an instance of *nastran-analysis-class*.

NASTRAN-ANALYSIS-CLASS [Class]

This class manages all communication between an instance of an *analysis-model-class* and the MSC Nastran software application, enables the writing of a bulk data file (deck), and enables the running of Nastran.

Properties

analysis-model-object	A references to an instances of an *analysis-model-class*. All nodes, elements, properties, load cases (boundary conditions), analysis types, and materials are obtained from the object specified in this property. (default nil)
analysis-directory	A string specifying the path to the directory where all files relating to this analysis will be written. Default is *(logical-path :nastran-data ^model-name)*
nastran-file-name	A string specifying the file name of the file (deck) in which the analysis data (cards) will be written. Default is *(format nil "~a.bdf" ^model-name)*
data-file	When demanded, this property will call a method that will demand all information necessary to pass to the external application and write the data file (deck) to the location specified with the *analysis-directory* and *nastran-file-name*.

run-nastran@	When demanded, this property will demand the *data-file* property and run Nastran using the command given in the *nastran-command* property with the specified data file (deck). The default location for the command is *(logical-path :nastran-path "nastran").*

Example

```
    (in-package :aml)


(define-class analysis-geometry-test-class
  :inherit-from (object)
  :properties (
               simple-beam-width  20.0
               simple-beam-height  1.0
               loaded-node-coords (list (/ ^simple-beam-width 2.0)
                                        (/ ^simple-beam-height -2.0)
                                        0.0)
               )
  :subobjects (
               (simple-beam :class '(tagging-object sheet-object)
                   width  ^^simple-beam-width
                   height ^^simple-beam-height
                   )
               (fixed-edge :class '(tagging-object line-object)
                   point1 (list (/ ^^simple-beam-width -2.0)
                                (/ ^^simple-beam-height -2.0)
                                0.0)
                   point2 (list (/ ^^simple-beam-width -2.0)
                                (/ ^^simple-beam-height 2.0)
                                0.0)
                   )
               (loaded-point :class 'point-object
```

```
                coordinates ^^loaded-node-coords
                )
            (imprinted-simple-beam :class 'imprint-class
                target-object    ^^simple-beam
                tool-object-list (list ^^fixed-edge  ^^loaded-point)
                )
            )
  )

(define-class analysis-mesh-test-class
  :inherit-from (object)
  :properties (
            geometry-model-object
            (default nil)
            (node-set :class 'analysis-node-set-class
         query-objects-list (list (the nodes-query (:from ^^simple-beam-mesh)))
                )
            )
  :subobjects (
            (mesh-database :class 'meshdb-class  )
            (simple-beam-mesh :class 'paver-mesh-class
                object-to-mesh (the imprinted-simple-beam
                                (:from ^^geometry-model-object))
                mesh-database-object ^^mesh-database
                element-shape  :hex
                solid-mesh?    nil
                )
            (simple-beam-elements :class 'mesh-elements-2d-query-class
                tagged-object-list (list
                           (the simple-beam (:from ^^geometry-model-object))
```

```
                                )
                mesh-object        ^^simple-beam-mesh
                )
             (fixed-nodes :class 'mesh-nodes-query-class
                tagged-object-list (list
                            (the fixed-edge  (:from ^^geometry-model-object))
                                )
                mesh-object        ^^simple-beam-mesh
                )
             (loaded-nodes :class 'mesh-query-nodes-from-interface-class
            interface-object (the loaded-point (:from ^^geometry-model-object)
                                )
                mesh-object      ^^simple-beam-mesh
                )
             )
  )

(define-class analysis-model-test-class
  :inherit-from (analysis-model-class)
  :properties (
               mesh-model-object          (default nil)
               z-load                    (default -100.0)
               geometry-model            (default nil)
               analysis-type             (default nil)
               load-case-objects-list    (list ^load-case-1)
               materials-list            (list 'steel)
               element-set-2d-objects-list  (list ^simple-beam-elements)
               property-set-objects-list    (list ^simple-beam-properties)
               material-catalog-object ^material-catalog
               mesh-object (the simple-beam-mesh (:from ^mesh-model-object))
```

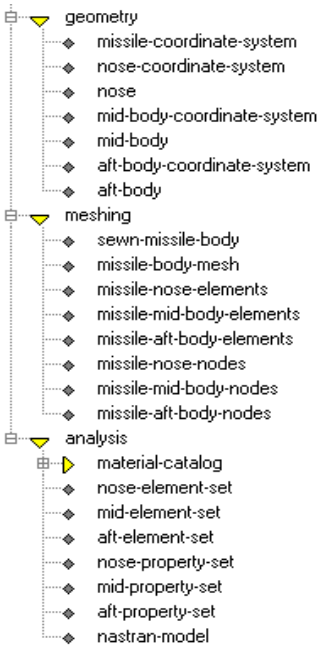
```
        node-set-objects-list (list (the node-set (:from ^mesh-model-object)))
              )
  :subobjects (
              (material-catalog :class 'material-catalog-class
                  )
              (simple-beam-properties
               :class 'analysis-property-set-2d-type-1-class
               material-name  "Steel"
               thickness      0.3
               )
              (simple-beam-elements
               :class 'analysis-element-set-2d-type-1-class
               query-objects-list (list
                  (the simple-beam-elements (:from ^^mesh-model-object)))
               property-set-object ^^simple-beam-properties
               )
              (fixed-nodes-constraint
               :class 'analysis-constraint-displacement-class
            target-object (the fixed-nodes  (:from ^^mesh-model-object))
               tx 0.0
               ty 0.0
               tz 0.0
               mx 0.0
               my 0.0
               mz 0.0
               )

              (nodal-load :class 'analysis-load-force-nodal-class
               target-object (the loaded-nodes
                              (:from ^^mesh-model-object))
```

```
                load-vector (list 0.0 0.0 ^^z-load)

                )

               (load-case-1

                :class 'analysis-load-case-class

                load-objects-list (list ^nodal-load)

                constraint-objects-list (list ^^fixed-nodes-constraint)

                )

               (nastran-interface :class 'nastran-analysis-class

                   analysis-model-object ^superior

                   nastran-file-name "SIMPLE-BEAM.bdf"

               nastran-version (nth 2 '(:nei-nastran :msc-nastran :nx-nastran))

                   )

               )

  )


(define-class post-processing-test-class

  :inherit-from (analysis-post-processing-structural-linear-static-nastran-
class)

  :properties (

               mesh-model-object     (default nil)

               analysis-model-object (default nil)

               mesh-database-object (the mesh-database

                                     (:from ^mesh-model-object))

               mesh-query-objects-list (list (the simple-beam-elements

                                              (:from ^mesh-model-object)))

               analysis-interface-nastran-object   (the nastran-interface
                                         (:from ^analysis-model-object))

               )

  )


(define-class analysis-test-class
```

```
:inherit-from (object)
:properties (
             )
:subobjects (
             (geometry-model :class 'analysis-geometry-test-class
                 )
             (mesh-model :class 'analysis-mesh-test-class
                 geometry-model-object ^^geometry-model
                 )
             (analysis :class 'analysis-model-test-class
                 analysis-type  :linear-static
                 mesh-model-object ^^mesh-model
                 )
             (post-processing :class 'post-processing-test-class
                 mesh-model-object     ^^mesh-model
                 analysis-model-object ^^analysis
                 )
             )
)
```

Exercise 3b

Given the above example of the AML Analysis Interface, create an analysis model for a modal analysis of the missile geometry and mesh given in Exercise 3a and add it as a subobject to the *missile-model-class*. The model tree hierarchy should resemble the following figure.



The steps needed to running the Missile Analysis Model are given below:

In AML:

- Create a model of the *missile-model-class*
- Inspect, draw, modify the model as needed
- Demand (the analysis nastran-model run-nastran@) property to run the Nastran analysis.

This will automatically demand the geometry, mesh and appropriate queries and create a files called nastran-model.xdb and nastran-model.bdf in the analysis directory. This file contains the results of the analysis.

Exercise 3b Solution

```
(in-package :aml)

(define-class missile-analysis-model-class
  :inherit-from (analysis-model-class)
  :properties (
           nose-mesh-query        nil
           mid-mesh-query         nil
           aft-mesh-query         nil
           mesh-object            nil


           ;;internal properties specific to the analysis-model-class
           property-set-objects-list (list
                     ^nose-property-set
                     ^mid-property-set
                     ^aft-property-set
                     )
           element-set-2d-objects-list (list
                       ^nose-element-set
                       ^mid-element-set
                       ^aft-element-set
                       )

           load-case-objects-list     nil

           material-catalog-object    ^material-catalog
           materials-list             (list 'steel)

           analysis-type              :modal
           )
  :subobjects (
              (material-catalog :class 'material-catalog-class
                )

           (nose-element-set :class 'analysis-element-set-2d-type-1-class
                 query-objects-list (list ^^nose-mesh-query)
                 property-set-object ^^nose-property-set
                 )

           (mid-element-set :class 'analysis-element-set-2d-type-1-class
                query-objects-list (list ^^mid-mesh-query)
                property-set-object ^^mid-property-set
                )

           (aft-element-set :class 'analysis-element-set-2d-type-1-class
                query-objects-list (list ^^aft-mesh-query)
                property-set-object ^^aft-property-set
                )

           (nose-property-set :class 'analysis-property-set-2d-type-1-class
                  material-name  "Steel"
                  thickness      0.3
                  )

           (mid-property-set :class 'analysis-property-set-2d-type-1-class
                 material-name  "Steel"
                 thickness      0.3
                 )

           (aft-property-set :class 'analysis-property-set-2d-type-1-class
                 material-name  'steel
```

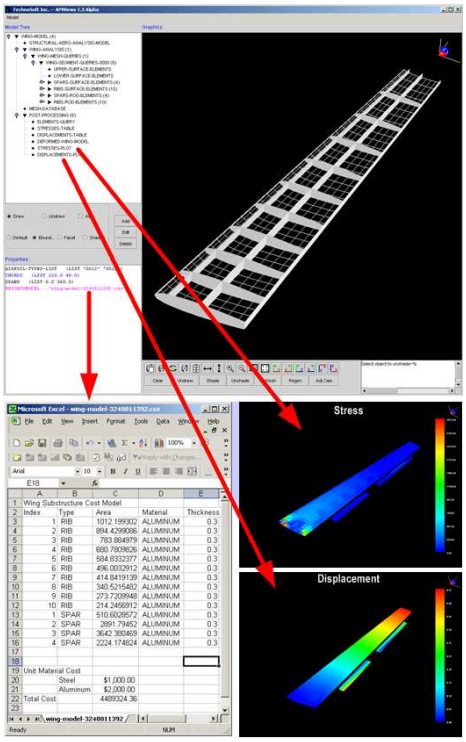
```
                thickness      0.3
                )

            (nastran-model :class 'nastran-analysis-class
                   analysis-model-object ^superior
                   )
                )
  )

(define-class missile-model-class
  :inherit-from (object)
  :properties (

           )
  :subobjects (

        (geometry :class 'missile-geometry-class

         )

       (meshing :class 'missile-mesh-model-class
         nose-object        (the nose (:from ^^geometry))
         mid-body-object    (the mid-body (:from ^^geometry))
         aft-body-object    (the aft-body (:from ^^geometry))
         )

       (analysis :class 'missile-analysis-model-class
         nose-mesh-query (the missile-nose-elements (:from ^^meshing))
         mid-mesh-query  (the missile-mid-body-elements (:from ^^meshing))
         aft-mesh-query  (the missile-aft-body-elements (:from ^^meshing))
         mesh-object     (the missile-body-mesh (:from ^^meshing))
         node-set-objects-list (list (the node-set (:from ^^meshing)))
         )
       (post-processing :class 'analysis-post-processing-structural-normal-
modes-nastran-class
         mesh-database-object (the mesh-database (:from ^^meshing))
         mesh-query-objects-list (list (the nose-mesh-query
                                       (:from ^^analysis))
                                     (the mid-mesh-query
                                       (:from ^^analysis))
                                     (the aft-mesh-query
                                       (:from ^^analysis)))
         analysis-interface-nastran-object (the nastran-model
                                             (:from ^^analysis))
         nodes-quantity (get-number-of-nodes (the missile-body-mesh
                             (:from ^^meshing)))
         )
    )
  )
```

Code Explanation

The analysis model is generated and demonstrated with a Nastran interface object. For example, an Ansys interface object could also be added very easily with a few lines of code, and the complete analysis could be performed in Ansys without having to redo the analysis model! Demanding the *run-nastran@* property will demand all of the geometry, mesh, mesh queries, materials, property sets, and element sets, needed to write the data file and run the analysis in Nastran. Instructions are given to post process the model in MSC Patran. AML also has a module for post processing including classes for extracting data from files, creating contour color plots, vector plots, graphs, and animations. These are not covered in this manual.

- ✓ Any change to the geometry or material properties (for example) will automatically smash the mesh and analysis objects/properties so that all dependencies are automatically managed. The model could now be used in conjunction with AML's optimization and design trade study analysis classes/tools to vary certain design variables to obtain optimized objectives or explore the design space.

8. Exporting and Visualizing AML models in XML format



AMEnterprise is a web-enabled environment for defining, managing, and publishing all aspects of an engineering project. It is an integrated suite consisting of: AMPublish, AMCentral, and AMViews, three integral modules for publishing, managing, and distributing complete product engineering data models. Geometric as well as non-geometric data relevant to product design details and associated processes can be released and distributed for viewing and inspection through a controlled-access environment. AMEnterprise facilitates collaboration among participants involved in a product engineering design.

AMCentral is a web application server providing project and user management functions. It controls the data flow among globally dispersed team members and application tools. AMCentral manages the releases of product and process data models. It provides key functionality for project data warehouse access control, version tracking, and data processing. AMCentral manages, processes, and catalogues product and process design changes. AMCentral provides key management and control functionality that include the ability to:

- Access part and process models and data over the web
- Link evaluation requests to associated applications and process the requested changes
- Provide notification of change requests and approvals to owners and clients (workflow)
- Manage the repository of project data, results, and design logs
- Manage models and data access privileges
- Track activity and usage on user and project levels

AMViews is an XML based browsing and inspection environment. It provides access to the AMCentral published products catalog to view and inspect available design alternatives from any remote location. The AMViews browser features a 3D interactive part geometry inspector and a customizable graphical user interface that can be exported from any AML-based application. With AMViews, product models, process data, and analysis/simulation results can be viewed and annotated, and alternative evaluation of parameters can be requested. AMViews key functionality includes:

- Viewing and annotations of product models and process data
- Requesting the evaluation of alternative design parameters
- Evaluation and comparison of published design and process data alternatives

AMPublish is a facility that allows the exporting of XML-based models from AML. It provides the facility to capture and publish trade studies and simulation results. It enables the documentation and annotation of a product and process design and provides the release of such documentation in an XML-based part model. The XML exporting facility within the AMPublish is customizable through a unique suite of AML based XML export methods. The AMCentral web application server facilitates the management of these models. AMViews enables the inspection and evaluation of the model alternative releases.

Portions of the AMPublish functionality can be accessed in AML through the *:aml-xml* and *:xml-parser* systems. Loading the *:aml-xml* system will automatically load the *:xml-parser* system. This section describes some of the functionality available to export AML models and supporting data in XML format and visualize them with AMViews. The general concept involves specifying a hierarchy of objects within a model and respective properties on those objects for export into an XML standard format. The system allows the exporting of objects, subobjects, properties, geometry, and supporting data such as data files (Ex. Excel, Word, Acrobat, PowerPoint, Visualizations, etc.) to a compact and portable format.

Methods & Properties to define to export XML [Methods and Properties]

The following constructs can be defined as methods on a class or defined as properties within a class to specify how the model will be exported. They are arranged with default behavior such that all objects will be exported with their respective geometry if available. For each object that is to be exported, add either a property or a method as given below:

Method/Property Name	Default Formula	Description
aml-xml-object-export?	`t`	Decides whether or not to export an object. This can be *t* or *nil*.
aml-xml-object-exported-subobjects	`t`	List of subobjects to be exported. Three options are available: *nil* => Do not write out any subobjects *t* => Go to each child and check the value returned from its *aml-xml-object-export?* property or method. List of subobjects => Instances to be exported
aml-xml-object-exported-attributes	`nil`	List of property names to export in symbol form.
aml-xml-object-export-geometry?	`t`	Decides whether or not to export an object's geometry. This can be *t* or *nil*.

The aforementioned methods/properties are most commonly defined. Other methods/properties are available as shown below. If not defined, they will be called using the default formula given in the table below.

Method/Property Name	Default Formula	Description
aml-xml-object-name	`(object-name self)`	Name of object to be written. This needs to be a symbol or a string.
aml-xml-object-type	`(type-of self)`	Type of object to be written. This needs to be a symbol or a string.
aml-xml-object-description	`(the label :error nil)`	Description of object that shows up as a tooltip for the object in the tree. This needs to be a string.
aml-xml-object-attribute-name	`(object-name attribute-object)`	Define this method to write the name of the attribute. This needs to be a symbol or a string.
aml-xml-object-attribute-value	`(the (:from attribute-object))`	Define this method to write the value of the attribute.
aml-xml-object-attribute-description	`(or (the aml-xml-object-description (:from attribute-object :error nil :relation nil)) (the label (:from attribute-object :error nil :relation nil)))`	Define this method to write the tooltip description of attribute. This needs to be a string.
aml-xml-object-get-geometry-ids	`(list (list (get-geom (the)) (the color) (the line-width) (the line-type) (the render)) )`	This is used to collect geometry for an object that will be written to the geometry file if *aml-xml-object-export-geometry-file?* is true. This is a list of lists, each list corresponding to geometric information on a geom. Each list is a list consisting of (geom-id color line-width line-type render). By default the method returns a list of a list of the object's: get-geom (or simple-geom geom), color, line-width, line-type, and render properties.

AML-XML-OBJECT-EXPORT [Method]

Call this method on the object that will form the root of the XML model. This will write an XML file (.xml) along with a geometry file (.xgl).

Format:

(aml-xml-object-export object xml-file-name
&key export-geometry?export-geometry-file-name aml-model-name
aml-model-file-name loaded-systems init-function)

Arguments:

Object	An instance that will form the root of the XML model.
xml-file-name	Full path and file name to the xml file. This should have a .xml extension.
export-geometry?	The default is set to *t*. If *t*, the method will write the .xgl geometry file.
export-geometry-file-name	The full path and file name to the geometry file. This defaults to "file.xgl" if *xml-file-name* is "file.xml".
aml-model-name	Name of the model (used only as an ID). This defaults to *nil*.
aml-model-file-name	AML model path name that should be loaded to recreate this XML model. This defaults to *nil*.
loaded-systems	AML systems to be loaded before loading this model. This defaults to *nil*.
init-function	Initialization function that should be called before retrieving the model. This defaults to *nil*.

Exporting Editable Attributes [Methods and Properties]

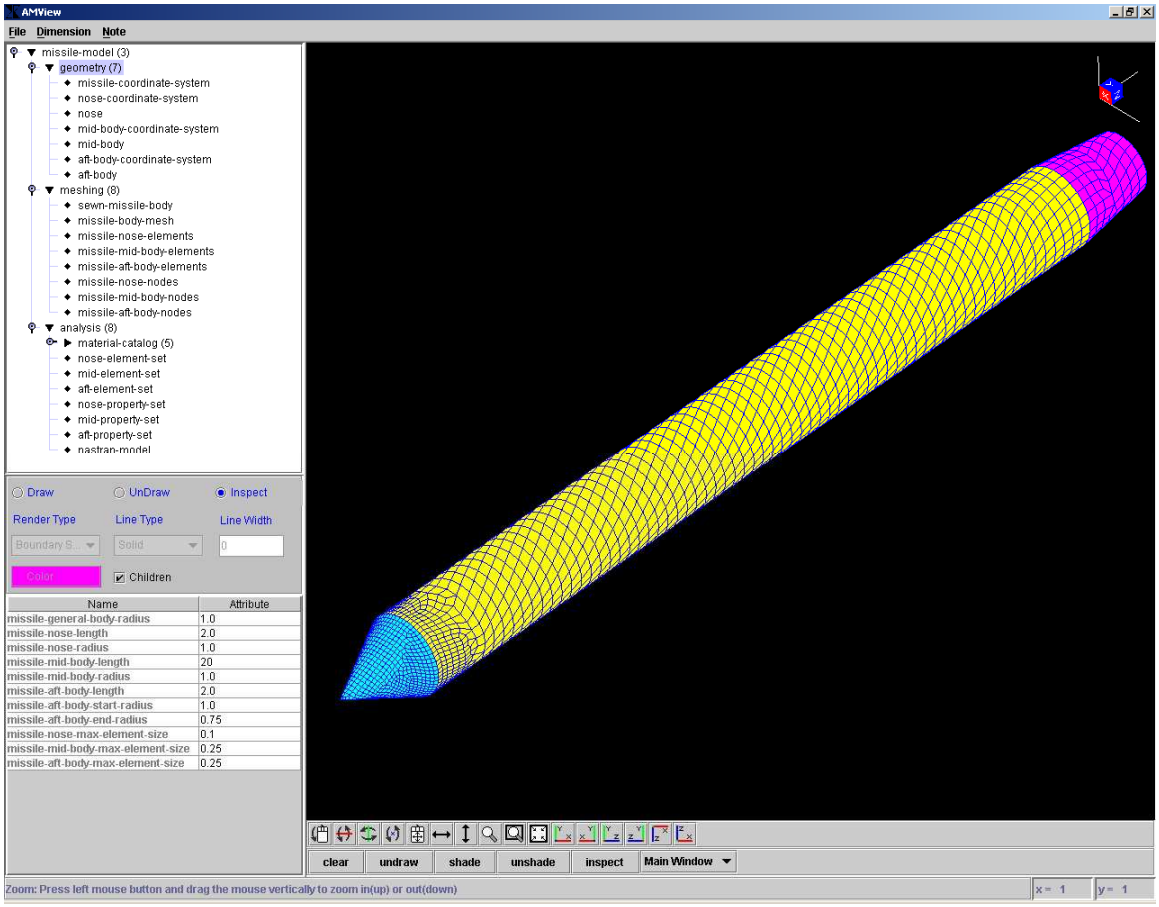
When visualized and annotated in AMviews, a model can be queried and modified for off-line collaboration and editing. If specified using the following methods or properties (written on property-objects), these exported properties/attributes can be changed by the user and saved into a new XML model. This new XML model can then be retrieved by an active AMcentral server, executed with the new configuration and modifications and re-saved with the new results obtained from the new configuration.

Method/Property Name	Default Formula	Description
aml-xml-object-attribute-type	`nil`	This will be used in the future for option-properties, application-types, etc.

aml-xml-object-attribute-editable?	`nil`	If set to *t*, it writes the attribute as editable, and a new value can be specified within AMviews for the attribute.
aml-xml-object-attribute-the-reference	`(find-tree (the))`	This is written for editable properties to enable the automatic update of an AML model with new values entered through AMviews

Exercise 11

Given the aforementioned methods, create an XML export model of the *missile-model-class* given in exercise 3b. This will include the geometry, mesh, and analysis models of the missile. The AMviews model hierarchy should resemble the following figure.



Export all of the geometry and objects in the hierarchy and specifically, export the following attributes from their respective classes:

1)**missile-geometry-class:** missile-general-body-radius, missile-nose-length, missile-nose-radius, missile-mid-body-length, missile-mid-body-radius, missile-aft-body-length, missile-aft-body-start-radius, missile-aft-body-end-radius, missile-nose-max-element-size, missile-mid-body-max-element-size, missile-aft-body-max-element-size
2)**patran-mesh-object:** object-to-mesh, element-shape, solid-mesh?
3)**analysis-property-set-class:** material-name, thickness

Create a method that exports the complete missile model for various configurations of the missile and view the resulting XML models in AMviews.

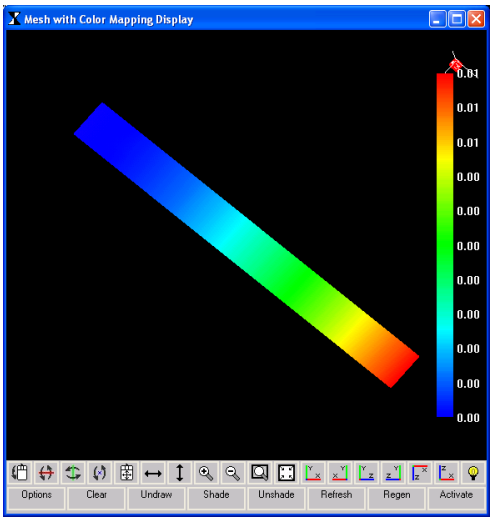
Exercise 11 Solution

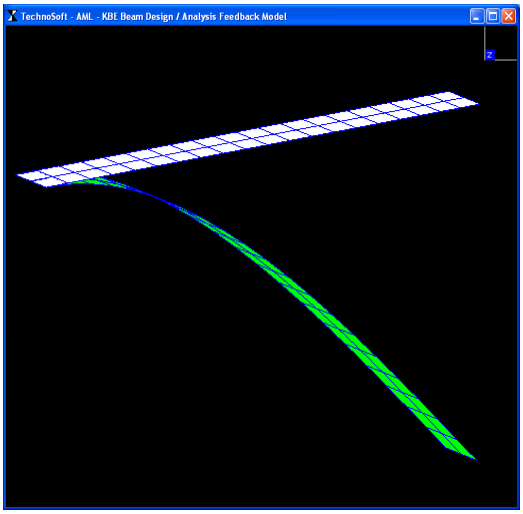
```
(in-package :aml)

;;(load-system :aml-xml)

(define-method
  aml-xml-object-exported-attributes
  missile-geometry-class ()
  '(
    missile-general-body-radius
    missile-nose-length
    missile-nose-radius
    missile-mid-body-length
    missile-mid-body-radius
    missile-aft-body-length
    missile-aft-body-start-radius
    missile-aft-body-end-radius
    missile-nose-max-element-size
    missile-mid-body-max-element-size
    missile-aft-body-max-element-size
    )
  )

(define-method
  aml-xml-object-exported-attributes
  analysis-property-set-class ()
  '(
    material-name
    thickness
    )
  )

(define-method
  aml-xml-object-exported-attributes
  patran-mesh-interface-class ()
  '(
    object-to-mesh
    element-shape
    solid-mesh?
    )
  )

(define-method
  missile-xml-export
  missile-model-class (directory file-name-prefix)
  (aml-xml-object-export
   self
   (format nil "~a.xml" (logical-path directory file-name-prefix))
   :export-geometry? t
   )
  )

(define-method
  missile-xml-export-configurations
  missile-model-class
 (directory file-name-prefix property-object-to-vary property-variance-list)
  (loop
    for i from 1
    for prop-value in property-variance-list
    do (change-property-value property-object-to-vary prop-value)
    (missile-xml-export self directory
                        (format nil "~a-~d" file-name-prefix i))
    )
  )
```

```
;;; (load-system :aml-xml)
;;; (create-model 'missile-model :class 'missile-model-class)
;;; (missile-xml-export-configurations
;;;  (the)
;;;  (logical-path :temp)
;;;  "missile"
;;;  (the geometry missile-mid-body-length self)
;;;  '(15 20 25)
;;;  )
```

Code Explanation

All of the objects in the AML model hierarchy are exported by default including the geometry for any object which has geometry. To make the XML model more informative, various properties and values from the geometry, mesh, and analysis are exported. These properties describe the configuration of the missile model at the time of the export, thus creating a configuration. Drawing the various objects in the hierarchy shows that geometry as well as mesh information was exported and stored for that configuration. The analysis was not run during the export process. This could have been accomplished simply by exporting the *run-nastran@* property from the *nastran-model* object. Additionally, the data file deck generated for the Nastran run could have been stored with the XML file for later use or verification. If AML was used for post processing of this model, those results could have also been stored in the XML file along with visualizations of the model. This is shown below in some sample screen shots.





9. Additional Useful AML Constructs

The following AML constructs are mentioned to give the user insight into a portion of important AML functions and methods not covered (or not covered in their entirety) in the AML Basic Training class. At this point, the trainee should be able to use the AML Reference Manual for further information on following AML constructs.

New AML Constructs

- *select-object* → query the model tree hierarchy
- *get-object* → interactively select an object from the screen
- *change-value* → change a property's value
- *change-formula* → change a property's formula
- *get-formula* → get a property's formula without demanding the property's value
- *format* → output to files, standard out, streams
- *with-open-file* → input/output to files
- *inheritance-list* → object's inheritance history list
- *object-name* → returns the object's name in symbol form
- *read-from-string* → takes a string and converts it to the corresponding AML entity
- debugging → functions such as *trace*, and using the * from the command prompt
- *describe* → runtime description of a function/method/class or an object
- *apropos* → query for a particular symbol in a function/method/class or an object defined in the current AML session
- *print-tree* → view the tree in the AML editor
- *find-tree* → returns the symbol representation of the *the* reference
- *trace-from* → places *the* reference at the specified object
- *i-depend-on* → list of objects/properties that depend on the specified object
- *i-affect* → list of objects/properties that affect on the specified object

10. After the AML Basic Training

10.1 Contacting TechnoSoft Inc.

The AML Basic Training course contains a considerable amount of information. Therefore the trainee may have questions while working on an application after the training. Questions specific to the following areas should be submitted to their respective email addresses.

Questions/Comments about:	**Email address:**
AML specific to the training manual and course	training-support@technosoft.com
Other AML functions/methods/classes	customer-support@technosoft.com
Bugs in AML functions/methods/classes	bugs@technosoft.com
AML reference manual (including undocumented functions/methods/classes)	manual@technosoft.com

10.2 Advanced Training Topics

The advanced training course is given on a user-specific basis depending on their intended application. The advanced course covers topics such as:

- advanced geometric classes and operations,
- the virtual geometry layer,
- advanced graphics and visualization,
- dimensioning and graphing,
- advanced user interface building,
- foreign function interactions,
- advanced debugging techniques,
- file input/output,
- data tables,
- writing advanced methods and functions,
- event classes,
- attribute tagging,
- meshing classes and querying,
- and other topics specific to the users' application.

11. Notes