Document Number: DSP0221

Date: 2011-10-18

Version: 3.0.0a

Managed Object Format (MOF)

Information for Work-in-Progress version:

This document is subject to change at any time without further notice.

It expires on: 04/15/2012

Provide any comments through the DMTF Feedback Portal:
http://www.dmtf.org/standards/feedback

IMPORTANT: This specification is not a standard. It does not necessarily reflect the views of the DMTF or all of its members. Because this document is a Work in Progress, this specification may still change, perhaps profoundly. This document is available for public review and comment until the stated expiration date.

Document Type: Specification

Document Status: Work in Progress - not a DMTF Standard

Document Language: en-US

Contents

Tables

Foreword

The CIM v3 Managed Object Format (MOF) (DSP0221) was prepared by the DMTF Architecture Working Group.

Versions marked as "DMTF Standard" are approved standards of the Distributed Management Task Force (DMTF).

DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability. For information about the DMTF see http://www.dmtf.org.

Acknowledgments

The DMTF acknowledges the following individuals for their contributions to this document:

Editors:

- George Ericson – EMC
- Wojtek Kozaczynski – Microsoft

Contributors:

- Andreas Maier – IBM
- Jim Davis – WBEM Solutions
- Karl Schopmeyer – Inova Development
- Lawrence Lamers – VMware
- Mike Brasher – Microsoft

Introduction

This document specifies the DMTF Managed Object Format (MOF), which is a definition language used to specify the interfaces of managed resources (storage, networking, compute, software) conformant with the CIM metamodel defined in DSP0004.

Comment [W.K.1]: Update the hyperlink to the final version

Typographical Conventions

The following typographical conventions are used in this document:

- Document titles are marked in *italics*.
- Important terms that are used for the first time are marked in *italics*.
- Examples are shown in the code blocks `code`.

Deprecated Material

Deprecated material is not recommended for use in new development efforts. Existing and new implementations may use this material, but they shall move to the favored approach as soon as possible. CIM services shall implement any deprecated elements as required by this document in order to achieve backwards compatibility. Although CIM clients may use deprecated elements, they are directed to use the favored elements instead.

Deprecated material should contain references to the last published version that included the deprecated material as normative material and to a description of the favored approach.

The following typographical convention indicates deprecated material:

DEPRECATED

Deprecated material appears here.

DEPRECATED

In places where this typographical convention cannot be used (for example, tables or figures), the "DEPRECATED" label is used alone.

Experimental Material

Experimental material has yet to receive sufficient review to satisfy the adoption requirements set forth by the DMTF. Experimental material is included in this document as an aid to implementers who are interested in likely future developments. Experimental material may change as implementation experience is gained. It is likely that experimental material will be included in an upcoming revision of the document. Until that time, experimental material is purely informational.

The following typographical convention indicates experimental material:

EXPERIMENTAL

Experimental material appears here.

EXPERIMENTAL

In places where this typographical convention cannot be used (for example, tables or figures), the "EXPERIMENTAL" label is used alone.

1 Scope

This document describes the syntax, semantics and the use of the Managed Object Format (MOF) for the DMTF Common Information Model (CIM) as defined in DSP0004 version 3.0.

The MOF provides the means to specify interface definitions of managed object types; including their properties, behavior and relationships with other objects. In the CIM context managed objects include logical concepts like policies, as well as real-world resource such as disk drives, network endpoints or software components.

MOF is used to define industry-standard object types, published by the DMTF as the CIM schema and other schemas, as well as user/vendor-defined object types that may or may not be derived from object types defined in schemas published by the DMTF.

This document does not describe specific CIM implementations, application programming interfaces (APIs), or communication protocols.

2 Normative references

The following documents are indispensable for the application of this document. For dated or versioned references, only the edition cited (including any corrigenda or DMTF update versions) applies. For references without a date or version the latest published edition of the referenced document (including any corrigenda or DMTF update versions) applies.

DMTF DSP0004, *Common Information Model (CIM) Infrastructure 3.0*
http://members.dmtf.org/apps/org/workgroup/technical/dmtf-arch/download.php/62435/DSP0004_3.0.0a_wipc2.docx

IETF RFC5234, *Augmented BNF for Syntax Specifications: ABNF*, January 2008
http://tools.ietf.org/html/rfc5234

ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards
http://isotc.iso.org/livelink/livelink.exe?func=ll&objId=4230456&objAction=browse&sort=subtype

ISO 639-1:2002, *Codes for the representation of names of languages – Part 1: Alpha-2 code*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22109

ISO 639-2:1998, *Codes for the representation of names of languages – Part 2: Alpha-3 code*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=4767

ISO 639-3:2007, *Codes for the representation of names of languages – Part 3: Alpha-3 code for comprehensive coverage of languages*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39534

ISO 3166-1:2006, *Codes for the representation of names of countries and their subdivisions – Part 1: Country codes*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39719

ISO 3166-2:2007, *Codes for the representation of names of countries and their subdivisions – Part 2: Country subdivision code*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39718

ISO 3166-3:1999, *Codes for the representation of names of countries and their subdivisions – Part 3: Code for formerly used names of countries*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=2130

ISO/IEC 10646:2003, *Information technology – Universal Multiple-Octet Coded Character Set (UCS)*
http://standards.iso.org/ittf/PubliclyAvailableStandards/c039921_ISO_IEC_10646_2003(E).zip

ISO/IEC 10646:2003/Amd 1:2005, *Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Amendment 1: Glagolitic, Coptic, Georgian and other characters*
http://standards.iso.org/ittf/PubliclyAvailableStandards/c040755_ISO_IEC_10646_2003_Amd_1_2005(E).zip

ISO/IEC 10646:2003/Amd 2:2006, *Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Amendment 2: N'Ko, Phags-pa, Phoenician and other characters*
http://standards.iso.org/ittf/PubliclyAvailableStandards/c041419_ISO_IEC_10646_2003_Amd_2_2006(E).zip

ISO/IEC 14750:1999, *Information technology – Open Distributed Processing – Interface Definition Language*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=25486

OMG, *Object Constraint Language*, Version 2.0
http://www.omg.org/cgi-bin/doc?formal/2006-05-01

OMG, *UML Superstructure Specification*, Version 2.1.1
http://www.omg.org/cgi-bin/doc?formal/07-02-05

The Unicode Consortium, *The Unicode Standard, Version 5.2.0, Annex #15: Unicode Normalization Forms*
http://www.unicode.org/reports/tr15/

3 Terms and definitions

Some terms used in this document have a specific meaning beyond the common English meaning. Those terms are defined in this clause.

The terms "shall" ("required"), "shall not," "should" ("recommended"), "should not" ("not recommended"), "may," "need not" ("not required"), "can" and "cannot" in this document are to be interpreted as described in ISO/IEC Directives, Part 2, Annex H. The terms in parenthesis are alternatives for the preceding terms, for use in exceptional cases when the preceding term cannot be used for linguistic reasons. Note that ISO/IEC Directives, Part 2, Annex H specifies additional alternatives. Occurrences of such additional alternatives shall be interpreted in their normal English meaning.

The terms "clause," "subclause," "paragraph," and "annex" in this document are to be interpreted as described in ISO/IEC Directives, Part 2, Clause 5.

The terms "normative" and "informative" in this document are to be interpreted as described in ISO/IEC Directives, Part 2, Clause 3. In this document, clauses, subclauses, or annexes labeled "(informative)" do not contain normative content. Notes and examples are always informative elements.

The Terms and Definitions defined in DSP0004 are included by reference.

3.1 Managed object format

Refers to the language described in this specification.

3.2 MOF file

Refers to a document with the content that conforms to the MOF syntax described by this specification.

3.3 MOF compilation unit

Refers to a set of MOF files, which includes those files that are explicitly listed as the input to the MOF compiler and the files directly or transitively included from the input files using the include pragma compiler directive.

3.4 MOF compiler

Program, which transforms a MOF compilation unit into another representation, such as an AST.

4 Symbols and abbreviated terms

The Symbols and Abbreviations defined in DSP0004 are included by reference.

4.1 MOF

Managed Object Format

4.2 ABNF

Augmented BNF

4.3 IDL

Interface definition language (http://en.wikipedia.org/wiki/Interface_description_language)

5 MOF file content

A MOF file contains MOF language statements, compiler directives and comments.

5.1 Encoding

The content of a MOF file shall be represented in Normalization Form C (http://www.unicode.org/reports/tr15/) and in the coded representation form UTF-8 (defined in ISO/IEC 10646:2003).

The content represented in UTF-8 should not have a signature sequence (EF BB BF, as defined in Annex H of ISO/IEC 10646:2003).

5.2 White space

White space in a MOF file is any combination of the following characters: space (U+0020), carriage return (U+000D), line feed (U+000A).

5.3 Line termination

The end of a line in a MOF file is indicated by one of the following:

- A sequence of carriage return (U+000D) followed by line feed (U+000A)
- A line feed (U+000A)
- Implicitly by the end of the MOF specification file, if the line is not ended by line end characters.

The different line end characters may be arbitrarily mixed within a single MOF file.

5.4 Comments

Comments in a MOF file do not create, modify, or annotate language elements. They shall be treated as white space by the MOF compilers and therefore are not represented in the language grammar.

Comments may appear anywhere in MOF syntax where white space is allowed and are indicated by either a leading double slash (//) or a pair of matching /* and */ character sequences.

A // comment is terminated by the end of line (see 5.3).

```
uint16 MyProperty;     // This is an example comment
```

A /* comment is terminated by the next */ sequence or by the end of the MOF file (whichever comes first).

```
uint16 MyProperty;     /* This is also
                               a comment */
```

5.5 Compiler directives

Compiler directives direct the processing of MOF files. Compiler directives do not create, modify, or annotate the language elements.

Compiler directives shall conform to the following structure

```
compiler Directive          =  "#pragma"  directiveName  "("  stringValue ")"
directiveName               =  IDENTIFIER
```

where IDENTIFIER is defined in Annex A, A.15.

The current standard compiler directives are listed in Table 1.

Table 1 – Standard Compiler Directives

Compiler Directive	Description
#pragma include(<path>)	This directive specifies that the referenced MOF specification file is included. The content of the referenced file shall be textually inserted at the point where the include directive is encountered. The path can be either an absolute file system path, or a relative path. If the path is relative, it is relative to the directory of the file with this the pragma. The content of the included field shall be one or more complete syntactic MOF production as defined in Annex A, A.1.

Comment [W.K.2]: Should we keep the ablsolute path.

What is the format of the relative path? Should it start with" .\" or just the name of sub-directory followed by "\ "

A MOF compiler may support additional compiler directives. Such new compiler directives are referred to as *vendor-specific compiler directives*. Vendor-specific compiler directives should have names that are unlikely to collide with the names of standard compiler directives defined in future versions of this specification. Future versions of this specification will not define compiler directives with names that begin with underscore (_, U+005F). Therefore, it is recommended that the names of vendor-specific compiler directives conform to the following structure:

```
directiveName = "_" org-id "_" IDENTIFIER
```

where org-id includes a copyrighted, trademarked, or otherwise unique name owned by the business entity that defines the compiler directive or that is a registered ID assigned to the business entity by a recognized global authority.

Vendor-specific compiler directives that are not understood by a MOF compiler shall be reported, but should be ignored by default. Thus, the use of vendor-specific compiler directives may affect the interoperability of MOF.

6 MOF language elements

MOF is an interface definition language (IDL) that is implementation language independent, and has syntax that should be familiar to programmers that have worked with other IDLs.

A MOF specification includes the following kinds of elements

- Compiler directives that direct the processing of the specification
- Qualifier declarations
- Type declarations such as classes, structures or enumerations, and
- Instances declarations.

Elements of MOF language are introduced and exemplified one-by-one in a sequence that progressively builds a meaningful MOF specification. To make the examples consistent we have created a small specification (a model) of a golf club membership. The root elements in the model are defined in the GOLF schema. All files of the model are listed in ANNEX C.

A complete description of the MOF syntax is provided in ANNEX A.

6.1 Qualifiers

A qualifier is a named and typed meta-data element. The value of a qualifier is associated with and describes a schema element. A detailed discussion of the qualifier concept and the list of standard qualifiers can be found in Section 7 of DSP0004.

A qualifier declaration in MOF corresponds to the QualifierType meta-model element, and is defined by the following ABNF rules:

```
qualifierDeclaration      =  [ qualifierList ] [TRANSLATABLE]
                             QUALIFIER  qualifierName ":" qualifierType qualifierScope [qualifierPolicy] ";"
qualifierName             =  IDENTIFIER
qualifierType             =  singleValueQualifierType / arrayValueQualifierType
singleValueQualifierType  =  ( primitiveType / enumName ) [ "=" defaultQualifierValue ]
arrayValueQualifierType   =  primitiveType "["  "]" [ "=" defaultQualifierValueList ]
defaultQualifierValue     =  literalValue
defaultQualifierValueList =  "{" primitiveTypeValueList  "}"
primitiveTypeValueList    =  literalPrimitiveTypeValue *( "," literalValue)
qualifierScope            =  "," SCOPE "(" ANY /  qualifiedElementList  ")"
qualifierPolicy           =  "," POLICY "(" APPLYMANY /  APPLYONCE / RESTRICTED ")"
qualifiedElementList      =  qualifiedElement  *(  ","  qualified Element )
qualifiedElement          =  STRUCTURE / CLASS / ASSOCIATION /
                             ENUMERATION  /  ENUMERATIONLITERAL /
```

Comment [W.K.3]: What is the default value if the policy is not specified? ApplyMany?

```
                              PROPERTY / REFPROPERTY /
                              METHOD / PARAMETER /
                              QUALIFIER
TRANSLATABLE          =  "translatable"          ; keyword: case insensitive
SCOPE                 =  "scope"                 ; keyword: case insensitive
ANY                   =  "any"                   ; keyword: case insensitive
POLICY                =  "policy"                ; keyword: case insensitive
APPLYMANY             =  "applymany"             ; keyword: case insensitive
APPLYONCE             =  "applyonce"             ; keyword: case insensitive
RESTRICTED            =   "restricted"           ; keyword: case insensitive
ENUMERATIONLITERAL    =  "enumerationliteral"    ; keyword: case insensitive
PROPERTY              =  "property"              ; keyword: case insensitive
REFPROPETY            =  "reference"             ; keyword: case insensitive
METHOD                =  "method"                ; keyword: case insensitive
PARAMETER             =  "parameter"             ; keyword: case insensitive
QUALIFIER             =  "qualifier"             ; keyword: case insensitive
```

The following MOF fragment defines the qualifier AggregationKind. The AggregationKind qualifier specifies the literals that define the kind of aggregation for a property with type reference. The type of the qualifier is a string enumeration with three literals; None, Shared and Composite.

```
Qualifier AggregationKind : CIM_AggregationKindEnum  = None,
   Scope(reference), Policy (applyMany);

enumeration CIM_AggregationKindEnum : string {
    None,
    Shared,
    Composite
};
```

Comment [W.K.4]: Should we follow the meta-model naming convention and spell the enum literals starting with lower-case letters?

A qualifier value in MOF represents an instance of the Qualifier meta-model element defined in DSP0004. The set of qualifier values specified in MOF on a schema element shall conform to the following qualifierList ABNF rule:

```
qualifierList                = "[" qualifier *( "," qualifier ) "]"
qualifier                    = qualifierName [ qualifierParameter ]
qualifierName                = IDENTIFIER
qualifierParameter           = qualifierValueInitializer / qualiferValueArrayInitializer
qualifierValueInitializer    = "(" literalPrimitiveTypeValue ")"
qualiferValueArrayInitializer = "{" primitiveTypeValue *( "," primitiveTypeValue ) "}"
```

The list of qualifier scopes (see the qualifiedElement rule above) includes the scope "qualifier", which implies that qualifier declarations can be qualified. The list of standard qualifiers is defined in DSP0004. Currently there are two standard qualifiers that can be specified on a qualifier declaration: Description and Deprecated. Note that there may be vendor-specific qualifiers with scope "qualifier".

6.2 Types

The data types in CIM are:

- Enumerations
- Structures
- Classes
- Associations, and
- Primitive types.

The subclauses below define how enumerations, structures, classes and associations are declared in MOF, and what primitive types are available in MOF.

Comment [W.K.5]: Changed “base type” to “primitive type” to make it consistant with the meta-model.

6.2.1 Enumeration declaration

There are two kinds of enumerations in CIM

- Integer enumerations, and
- String enumerations.

CIM integer enumerations are comparable to enumerations in programming languages; each enumeration literal in an integer enumeration is represented by a distinct integer value.

In string enumerations, which can be found in UML and are similar to XML enumerations, each enumeration literal is represented by a distinct string value.

An enumeration declaration in MOF corresponds to the Enumeration meta-model element defined in DSP0004 and shall conform to the following enumDeclaration ABNF rule:

```
enumDeclaration = [ qualifierList ] [FINAL]
                  ENUMERATION enumName ":"
                  ( integerEnumDeclaration /
                    stringEnumDeclaration /
                    derivedEnumDeclaration )
enumName        = ( IDENTIFIER  / schemaQualifiedName )
```

```
integerEnumDeclaration   =  DT_UnsignedInteger "{" integerEnumElement *( "," integerEnumElement) "}"  ";"
stringEnumDeclaration    =  DT_STRING  "{" stringEnumElement *( "," stringEnumElement) "}"  ";"
integerEnumElement       =  [ qualifierList ] enumLiteral "=" integerValue
stringEnumElement        =  [ qualifierList ] enumLiteral [ "=" stringValue ]
derivedEnumDeclaration   =  ( IDENTIFIER  / schemaQualifiedName )
                            "{" enumElement *("," enumElement ) "}"  ";"
enumElement              =  enumLiteral [ "=" (integerValue / stringValue) ] ";"
enumLiteral              =  IDENTIFIER
ENUMERATION              =  "enumeration"              ; keyword: case insensitive
```

Comment [W.K.6]: Modeled the values of integer enumerations non-optional.

Comment [W.K.7]: Left string values optional.

The integerEnumElement rule states that integer enumeration elements must have explicit integer velues. Those values must be unique and escending. The stringEnumElement rule states that the values of string enumeration elements are optional. If not declared, the value of a string enumeration element is assumed to be the same as it literal. This is illustrated in the the examples below.

There golf club model contains a number of enumeration declarations. Enumerations can be defined at the schema level or inside declarations of structres, classes or associations. Enumerations defined inside those other types are referred to as “embedded” enumeration declarations. The names of schema-level enumerations must conform to the schemaQualifiedName MOF rule that is, their names must start with the name of the schem followed by the underscore (U+005F).

The schema-level string enumeration GOLF_MonthsEnum shown below defines months of the year.

```
enumeration GOLF_MonthsEnum : string {
    January,
    February,
    March,
    April,
    May,
    June,
    July,
    August,
    September,
    October,
    November,
    December
};
```

As explained above, string enumerations don’t require assigning values to their literals. If a value is not assigned to a literal, it is assumed that it identical to the liter itself, so for example the value of the literal January above is “January”.

The GOLF_StatesEnum is an example of another schema-level string enumeration that assigns values to all of its literals, which are different then the literal names.

```
enumeration GOLF_StatesEnum : string {
```

```
    AL = "Alabama",
    AK = "Alska",
    AZ = "Arizona",
    AR = "Arkansas",
    CA = "California",
    CO = "Colorado",
    CT = "Connecticut",
    DE = "Delaware",
    FL = "Florida",
    GA = "Georgia",
    HI = "Hawaii",
    ID = "Idaho",
    IL = "Illinois",
    IN = "Indiana",
    IA = "Iowa",
    KS = "Kansas",
    LA = "Louisiana",
    ME = "Main",
    MD = "Maryland",
    MA = "Massachusetts",
    MI = "Michigan",
    MS = "Mississippi",
    MO = "Missouri",
    MT = "Montana",
    NE = "Nebraska",
    NV = "Nevada",
    NH = "New Hampshire",
    NJ = "New Jersey",
    NM = "New Mexico",
    NY = "New York",
    NC = "North Carolina",
    ND = "North Dakota",
    OH = "Ohio",
    OK = "Oklahoma",
    OR = "Oregon",
    PA = "Pennsylvania",
    RI = "Rhode Island",
    SC = "South Carolina",
    SD = "South Dakota",
    TX = "Texas",
    UT = "Utah",
    VT = "Vermont",
    VA = "Virginia",
    WA = "Washington",
    WV = "West Virginia",
    WI = "Wisconsin",
    WY = "Wyoming"
};
```

The following two enumerations are examples of embedded integer enumerations that derive from each other. The declaration of the MemberStatusEnum is embedded in the declaration of the GOLF_ClubMember class. It defines three elements and assigns integer values to them. The assignment of values to the integer enumeration literals is not optional (see the integerEnumElement rule). The values shall to be assigned in ascending order, but don't have to be consecutive.

```
  [Description (
        "Instances of this class represent members of a golf club." )]
class GOLF_ClubMember: GOLF_Base {

// ====================== embedded enumerations ======================
    enumeration MemberStatusEnum : Uint16 {
        Basic = 0,
        Extended = 1,
        VP = 2
    };

// ============================ properties ============================
    MemberStatusEnum Status;
    GOLF_Date MembershipEstablishedDate;

    ...
};
```

The ProfessionalStatusEnum enumeration is defined in the GOLF_Professional class that inherits from the GOLF_ClubMember class. It derives from the MemberStatusEnum enumeration declared in the parent class and adds two elements to it. The elements are not given consecutive values (the values of the MemberStatusEnum end at 2 and the values of ProfessionalStatusEnum start at 6). This is intent ended to leave the range 3, 4 and 5 for elements that can be added to the MemberStatusEnum enumeration at later time.

```
class GOLF_Professional : GOLF_ClubMember {
// ======================== embedded structures ========================
    ...

// ====================== embedded enumerations ======================
    enumeration ProfessionalStatusEnum : MemberStatusEnum {
        Professional = 6,
        SponsoredProfessional = 7
    };

// ============================ properties ============================
    ...
};
```

Declarations of CIM enumerations should meet the following model integrity constraints

Comment [W.K.8]: Not the final list

- Only qualifiers with the scope "enumeration" can be applied to enumeration declarations

- A derived enumeration can only add literals
- Enumeration literals shall be unique in an enumeration
- A derived enumeration is of the same primitive type as its base enumeration and therefore the values assigned to its literals shall be of that type
- The elements in a derived enumeration of integer primitive type shall have values
- The definition order of *integer* enumeration literals is significant and their values shall increase
- The definition order of the *string* enumeration elements does not matter
- …

6.2.2 Structure declaration

A CIM structure defines a complex type that has no independent identity, but can be used as a type of a property, a method result or a method parameter.

The structure declaration in MOF corresponds to the Structure meta-model element element defined in DSP0004 and shall conform to the structureDeclaration ABNF rule:.

```
structureDeclaration  =  [ qualifierList ] [FINAL]
                         STRUCTURE structureName [superStructure]
                         "{" *propertyDeclaration "}" ";"
structureName         =  ( IDENTIFIER  / schemaQualifiedName )
superStructure        =  ":" ( IDENTIFIER / schemaQualifiedName )
FINAL                 =  “final”                    ; keyword: case insensitive
STRUCTURE             =  "structure"                ; keyword: case insensitive
```

Structure is a, possibly empty, collection of properties that can derive from another structure (see the *general* association in CIM meta-model in DSP004) . A structure can be declared at the schema level, and therefore be visible to all other structures, classes and associations in the schema, or its declaration can be embedded in a class declaration and be visible only to that class and its ancestors.

A property declaration in MOF (for use by structures, classes and associations) corresponds to the Property meta-model element defined in DSP0004 and shall conform to the propertyDeclaration ABNF rule:

```
propertyDeclaration   =  [ qualifierList ] [KEY]
                         ( basePropertyDeclaration /
                           structurePropertyDeclaration /
                           enumPropertyDeclaration /
                           classPropertyDeclaration /
                           refererecePropertyDeclaration )
basePropertyDeclaration = primitiveType propertyName [ array ] [ "=" primitiveTypeDefaultValue ] ";"
```

Comment [W.K.9]: The meta-model introduces two meta-properties of properties
- key, and
- static

I have only added the key, and would like to disucss the static on separate thread.

```
structurePropertyDeclaration   =  structureName propertyName [ array ] [  "=" structureDefaultValue ] ";"
enumPropertyDeclaration        =  enumName propertyName [ array ] [  "=" enumDefaultValue ] ";"
classPropertyDeclaration       =  className propertyName [ array ] ["=" structureDefaultValue ] ";"
referencePropertyDeclaration   =  className REF propertyName [ array ] [ "=" referenceDefaultValue ] ";"
array                          =  "[" "]"
propertyName                   =  IDENTIFIER
KEY                            =  "key"                    ; keyword: case insensitive
REF                            =  "ref"                    ; keyword: case insensitive
```

Comment [W.K.10]: We have removed the array size. Array size constraints should be specified using the Constraint qualifier.

The GOLF_Date is an example of a schema-level structure with three properties.

```
structure GOLF_Date {
    UInt16 Year = 2000;
    GOLF_MonthsEnum Month = GOLF_MonthsEnum.January;
        [MinValue(1), MaxValue(31)]
    UInt16 Day = 1;
}
```

All of the properties have default values that set the structure value to January 1, 2000 if no other values are provided. The default value of the Month property can be simplified to the form

```
GOLF_MonthsEnum Month = January
```

as the enumeration type is implied by the property declaration. The Day property has the min and max values defined using the MinValue and MaxValue qualifiers.

The use of the GOLF_Date structure as the type of a property is shown in previous section in the declaration of the GOLF_ClubMember class; the property is called MembershipEstablishedDate.

The Sponsor structure, see below, is defined inside the GOLF_Professional class. It is an example of an embedded structure declaration, which can be used only in the class it is defined in, or a derived class.

```
class GOLF_Professional : GOLF_ClubMember {
// ======================== embedded structures ========================
    structure Sponsor {
        string Name,
        GOLF_Date ContractSignedDate;
        Real32 ContractAmount;
    };

// ======================== embedded enumerations ========================
```

```
    enumeration ProfessionalStatusEnum : MemberStatusEnum {
        Professional = 6,
        SponsoredProfessional = 7
    };

// ============================ properties ============================
        [Override]
    ProfessionalStatus Status = Professional;
    Sponsor Sponsors[];
    Boolean Ranked;
};
```

The declaration of the default value of the Status property is an example of short version of the ProfessionalStatus.Professional enumeration element value.

A declaration of a CIM structure should conform to the following model integrity constraints

Comment [W.K.11]: Not the final list

- Only qualifiers with the scope “structure” shall be applied to a structure declaration
- An embedded structure can inherit from a schema-level structure, but a schema-level structure shall not inherit from an embedded structure
- Property names in a structure shall be unique
- Overriding property shall not change the property type
- A structure shall not have a key property (a property with the modifier “key”)
- Only qualifiers with the scope “property” shall be applied to property declaration
- Scalar properties should have single default value
- Array properties should have array default values, even if that array contains only a single element
-
- …

6.2.3 Class declaration

A class defines both properties and methods (the behavior) of its instances, which have unique identity in the scope of a server, a namespace, and the class. A class may define embedded structures and enumerations.

A class declaration in MOF corresponds to the Class meta-model element defined in DSP0004 and shall conform to the classDeclaration ABNF rule:

```
classDeclaration    =  [ qualifierList ] *2classModifier
                       [ABSTRACT] CLASS className [ superClass ]
                       "{" *classFeature "}" ";"
classModifier       =  ABSTRACT / FINAL
className           =  schemaQualifiedName
superClass          =  ":" schemaQualifiedName
classFeature        =  propertyDeclaration /
                       enumerationDeclaration /
```

Comment [W.K.12]: This rule allows the write the class modifiers in any order; abstract final, or final abstract.
Should we change require a fixed order?

```
                                structureDeclaration /
                                methodDeclaration
ABSTRACT                =   "abstract"                  ; keyword: case insensitive
CLASS                   =   "class"                     ; keyword: case insensitive
```

The embedded enumeration and structure declarations have been introduced in sections 6.2.1 and 6.2.2 respectively. Section 6.2.2 also introduced the property declaration.

A method declaration within a class declaration in MOF corresponds to the Method meta-model element defined in DSP0004 and shall conform to the methodDeclaration ABNF rule:

```
methodDeclaration       =   [ qualifierList ] [ STATIC ] ( dataType [ array ] / VOID )  methodName
                                "(" [ parameterList ] ")" ";"
methodName              =   IDENTIFIER
VOID                    =   "void"                      ; keyword: case insensitive
STATIC                  =   “static”                    ; keyword: case insensitive
parameterList           =   parameterDeclaration *( "," parameterDeclaration )

parameterDeclaration    =   [ qualifierList ] [ IN / OUT / INOUT ]
                                ( primitiveParamDeclaration /
                                  structureParamDeclaration /
                                  enumParamDeclararion /
                                  classParamDeclararion /
                                  referenceParamDeclaration )
primitiveParamDeclaration   =   primitiveType parameterName [ array ] [ "="  primitiveTypeDefaultValue ]
structureParamDeclartion    =   structureName parameterName [ array ] ["=" structureDefaultValue ]
enumParamDeclaration        =   enumName parameterName [ array ]    ["=" enumDefaultValue ]
classParamDeclaration       =   className parameterName [ array ] ["="structureDefaultValue]
referenceParamDeclaration   =   className "REF" parameterName [ array ] [ "=" referenceDefaultValue ]
parameterName           =   IDENTIFIER
IN                      =   "in"                        ; keyword: case insensitive
OUT                     =   "out"                       ; keyword: case insensitive
INOUT                   =   "inout"                     ; keyword: case insensitive
```

A class may define two kinds of methods

- Instance methods, which are invoked on an instance of the class (the concept is similar to the "this" method argument in dynamic programming languages), and
- Static methods, which are invoked on the class.

The following intrinsic methods are predefined on each class, that is, the user does not have to specify them in MOF but the class provider should implement them:

- Intrinsic instance methods
  - Get
  - Delete
  - Modify
- Intrinsic static methods
  - Enumerate
  - Create.

A class can derive from another class, in which case it inherits the enumerations, structures, properties and methods of its superclass. A class can also derive from a structure, in which case it only inherits the properties of the parent structure.

Below are the declarations of three central classes of our GOLF schema example; GOLF_Base, GOLF_Club and GOLF_ClubMember.

```
// ===================================================================
//  GOLF_Base
// ===================================================================
abstract class GOLF_Base {
    [Description (
       "InstanceID is a property that opaquely and uniquely identifies "
           "an instance of a class that derives from the GOLF_Base structure. ")]
  key string InstanceID;

    [Description (
       "A a short textual description (one- line string) of the "
           "instance." ),
     MaxLen (64)]
  string Caption;
};

// ===================================================================
//  GOLF_Club
// ===================================================================
  [Description (
          "Instances of this class represent golf clubs. A golf club is an "
          "an organization that provides memebr services to golf players "
          "both amateur and professional." )]
class GOLF_Club: GOLF_Base {
// ============================ properties ============================
```

Comment [W.K.13]: Replaced [Key] qualifier with *key* keyword.

```
    string ClubName;
    GOLF_Date YearEstablished;

    GOLF_Address ClubAddress;
    GOLF_Phone ClubPhoneNo;
    GOLF_Fax ClubFaxNo;
    string ClubWebSiteURL;

    GOLF_ClubMember REF AllMembers[];
    GOLF_Professional REF Professionals[];

// ============================ methods ============================
    static GOLF_ResultCodeEnum AddNonProfessionalMember (
        in GOLF_ClubMember newMember
    );

    static GOLF_ResultCodeEnum AddProfessional (
        in GOLF_Professional newProfessional
    );

    static UInt32 GetMembersWithOutstandingFees (
        in GOLF_Date referenceDate,
        out GOLF_ClubMember REF lateMembers[]
    );

    static GOLF_ResultCodeEnum TerminateMembership (
        in GOLF_ClubMember REF member
    );
};

// ==================================================================
//  GOLF_ClubMember
// ==================================================================
  [Description (
        "Intsances of this class represent members of a golf club." )]
class GOLF_ClubMember: GOLF_Base {

// ====================== embeded enumerations ======================
    enumeration MemberStatusEnum : Uint16 {
        Basic = 0,
        Extended = 1,
        VP = 2
    };

// ============================ properties ============================
    string FirstName;
    string LastName;
    MemberStatusEnum Status;
```

```
    GOLF_Date MembershipEstablishedDate;

    Real32 MembershipSignUpFee;
    Real32 MonthlyFee;
    GOLF_Date LastPaymentDate;

    GOLF_Address MemberAddress;
    GOLF_PhoneNumber MemberPhoneNo;
    string MemberEmailAddress;

// ============================ methods ============================
    GOLF_ResultCodeEnum SendPaymentReminderMessage();
};
```

The GOLF_Base class is the base from which all other GOLF schema classes derive. It is an abstract class and it introduces a key property. A key property should be of type string, although other primitive types can be used, and must have the key modifier keyword. The key property is used by class implementations in servers to uniquely identify instances of the class in a namespace. The clients should make no assumptions about its content when instance paths are returned by the server. In other words the class implementation in the server sets the key property to a value that allows it to uniquely identify the represented managed object when it gets the instance path back from the client. CIM v3 makes an assumption that clients should not construct instance paths and only use the paths returned by the server.

Comment [W.K.14]: This deserves a separate discussion in the meta-model document.

All methods in the GOLF schema classes return the same result type, which is the enumeration GOLF_ResultCodeEnum defining the DMTF stand result codes (see *Annex C* for the definition).

All methods of the GOLF_Club class are static and three of them manage two light-weight associations represented by the properties AllMembers and Professionals. Those two properties are instance reference arrays and model one-way associations from the GOLF_Club to GOLF_ClubMember and GOLF_Professional respectively. The static methods perform the following operations.

AddNonProfessionalMember	Adds new non-professional member to the club and adds that member to the AllMembers association. Notice that the design of the method suggests that the implementation of the method will have to call CreateInstance intrinsic method on the GOLF_ClubMember class to create a new member instance and get a reference to it.
AddProfessional	Adds new professional member to the club and adds that member to both the Professionals and the AllMembers associations. As above, the design implies that it calls CreateInstance intrinsic method on the GOLF_Professional class to create an instance of the professional member.
TerminateMembership	Terminates the club membership and removes the member from AllMembers and possibly Professionals associations, depending on the membership status. Notice that in this case the method gets a reference to the existing member instance as an input parameter.

GetMembersWithOutstandingFees	Finds club members that have not paid the club dues on time. This method returns references to the “late” members in the form of a reference array.

The GOLF_ClubMember class has one extrinsic instance method SendPaymentReminderMessage. The method has no explicit parameters as it takes an instance of GOLF_ClubMember as its default argument. The design suggests that this method is called on all members that are returned by the method GetMembersWithOutstandingFees described above.

CIM v3 introduces the ability to define default values for method parameters (see the primitiveParamDeclaration, structureParamDeclararion, enumParamDeclaration, classParamDeclaration and referenceParamDeclaration MOF grammar rules). An example is the GetNumberOfProfessionals method in the GOLF_Professional class.

```
// ===================================================================
//  GOLF_Professional
// ===================================================================
class GOLF_Professional : GOLF_ClubMember {
// ======================= embeded structures =======================
      structure Sponsor {
            string Name,
            GOLF_Date ContractSignedDate;
            Real32 ContractAmount;
      };

// ======================= embeded enumerations =====================
      enumeration ProfessionalStatusEnum : MemberStatusEnum {
            Professional = 6,
            SponsoredProfessional = 7
      };

// ============================ properties ===========================
            [Override]
      ProfessionalStatusEnum Status = Professional;
      Sponsor Sponsors[];
      Boolean Ranked;

// ============================= methods =============================
      static GOLF_ResultCodeEnum GetNumberOfProfessionals (
            out Uint32 NoOfPros,
            in ProfessionalStatusEnum = Professional
      )
};
```

The second parameter of the method has the default value MemberStatusEnum.Professional. The parameter default values have been introduced to support the so-called method extensions. The idea is as flows

- A derived class may override a method and add a new input parameter
- The added parameter is declared with a default value
- A client written agains the base class calls the method without that parameter because it does not know about it
- The class implementation does not error out, but takes the default value of the missing parameter and executes the “extended” method implementation.

The example does not illustrate method overriding to keep it simple. However the GetNumberOfProfessionals method can be called with two arguments, or only with one out argument.

The same mechanism can be used when upgrading a schema, where clients written against a previous version can call extended method in a new version.

Method parameters are identified by name and not by position. Therefore parameters with default values can be added to the method signature at any position, and clients invoking the method can pass arguments in any order.

Declarations of a CIM class must conform to the following model integrity constraints

Comment [W.K.15]: Not the final list

- Only qualifiers with the scope including “class” shall be applied to class declarations
- Classes shall inherit from classes or structures
- Method names in a class shall be unique
- Property names in a class shall be unique
- An abstract class shall not inherit from a concrete class
- Concrete class shall have a key property
- There can be only one key property in a class
- Only qualifiers with the scope including “method” shall be applied to method declarations
- Only qualifiers with the scope including “parameter” shall be applied to method parameters
- Method parameters with default value
  - must form a consecutive trailing group
  - must have the `in` modifier keyword
- …

6.2.4 Association declaration

An association declaration in MOF corresponds to the Association meta-model element defined in DSP0004 and shall conform to the following associationDeclaration ABNF rule:

```
associationDeclaration  =  [ qualifierList ] *2classModifier
                           ASSOCIATION associationName [ superAssociation ]
                           "{" * classFeature "}" ";"
associationName         =  schemaQualifiedName
superAssociation        =  ":" schemaQualifiedName
ASSOCIATION             =  "association"              ; keyword: case insensitive
```

In the meta-model the Association derives from Class. The Association is structurally identical to Class, but its declaration

- must have a least two, scalar value, reference properties (inherited and defined in the association), and
- the set of reference properties of the association (inherited and defined in the association) represent the ends of the association.

The GOLF_MemberLocker below is an example of an association with two ends and it represents an assignment of lockers to club members.

```
// ==================================================================
//  GOLF_MemberLocker
// ==================================================================
association GOLF_MemberLocker : GOLF_Base {
        [Max(1)]
    GOLF_Member REF Member;
        [Max(1)]
    GOLF_Locker REF Locker;
    GOLF_Date AssignedOnDate;
};
```

Notice that the GOLF_MemberLocker association derives from GOLF_Base, which defines the key property InstanceID. This is a change to the convention established for CIM v2 schemas. In CIM v2 schemas, the key of an association instance was usually the combined value of its reference properties. In CIM v3 schemas, it is recommended that association instances are identified the same way as instances of ordinary classes using a single, client-opaque key property.

Also notice that the multiplicity of the association's ends can be defined using the Max and Min qualifiers (see the discussion of associations in Section 6.4 of DSP0004).

A declaration of a CIM association must meet the following integrity constraints:

- Only qualifiers with the scope including "association" shall be applied to association declarations
- An association can inherit from an association, a structure, and form a class with no reference properties
- Other class constrains apply to associations
- An association must have at least two reference properties
- …

Comment [W.K.16]: This is a sensible engineering constraint, but we need to verify if we should allow turning classes to associations.

6.2.5 CIM Primitive types declarations

CIM has the following set of primitive data types. Each MOF primitive data type corresponds to a meta-model element derived from the PrimitiveType element defined in DSP0004.

- Numeric primitieve types
  - Integers
  - Reals

- string
- datetime
- boolean, and
- octetstring.

A MOF base data type shall conform to the primitiveType ABNF rule:

```
primitiveType       = DT_Integer /
                      DT_REAL32 /
                      DT_REAL64 /
                      DT_STRING /
                      DT_DATETIME /
                      DT_BOOLEAN /
                      DT_OCTETSTRING
DT_Integer          = DT_UnsignedInteger /
                      DT_SignedInteger
DT_UnsignedInteger  = DT_UINT8 /
                      DT_UINT16 /
                      DT_UINT32 /
                      DT_UINT64
                      DT_UINT128
DT_SignedInteger    = DT_SINT8 /
                      DT_SINT16 /
                      DT_SINT32 /
                      DT_SINT64 /
                      DT_SINT128 /
DT_UINT8            = "uint8"           ; keyword: case insensitive
DT_UINT16           = "uint16"          ; keyword: case insensitive
DT_UINT32           = "uint32"          ; keyword: case insensitive
DT_UINT64           = "uint64"          ; keyword: case insensitive
DT_UINT128          = "uint128"         ; keyword: case insensitive
DT_SINT8            = "sint8"           ; keyword: case insensitive
DT_SINT16           = "sint16"          ; keyword: case insensitive
DT_SINT32           = "sint32"          ; keyword: case insensitive
DT_SINT64           = "sint64"          ; keyword: case insensitive
DT_SINT128          = "sint128"         ; keyword: case insensitive
```

```
DT_REAL32          =  "real32"          ; keyword: case insensitive
DT_REAL64          =  "real64"          ; keyword: case insensitive
DT_DATETIME        =  "datetime"        ; keyword: case insensitive
DT_STRING          =  “string”          ; keyword: case insensitive
DT_BOOLEAN         =  “boolean”         ; keyword: case insensitive
DT_OCTETSTRING     =  “octetstring”     ; keyword: case insensitive
```

The primitive types are used in the declaration of

- Qualifiers
- Properties
- Enumerations
- Method parameters, and
- Method results.

6.3 Instance and value definitions

The CIM meta-model, and therefore MOF, allow for defining instances of classes and associations, and values of structures. The instanceDeclaration in MOF maps to the InstanceDeclaration in the meta-model, and its ABNF rules are as follows:

```
instanceDeclaration    =  INSTANCE OF ( structureName / className / associationName )  [ alias ]
                          propertyValueList ";"
propertyValueList      =  "{" *propertyValue "}"
propertyValue          =  propertyName "=" value ";"
propertyName           =  IDENTIFIER
value                  =  singleValue /
                          valueArray /
singleValue            =  simpleValue /
                          complexValue
valueArray             =  "{" [ valueArrayElementList  ] "}"
valueArrayElementList  =  singleValue *("," singleValue )
simpleValue            =  literalPrimitiveTypeValue /
                          literalObjectPathValue /
                          aliasIdentifier
complexValue           =  propertyValueList /
                          instanceDeclaration
```

Comment [W.K.17]: Should we introduce rules to define VALUE OF structureName [ALIAS] ...?

Comment [W.K.18]: This is called Instance in the meta-model

Comment [W.K.19]: The value could be NULL

Comment [W.K.20]: This is questioned; should we allow “instance of X { ...” to the right of the property name

```
alias          =  AS aliasIdentifier
INSTANCE       =  "instance"          ; keyword: case insensitive
AS             =  "as"                ; keyword: case insensitive
OF             =  "of"                ; keyword: case insensitive
```

The example below is a definition of an instance of GOLF_ClubMember, which represents a person with the name John Doe.

Values of structures can be defined in two ways

- By inligning them inside the owner class or structure instance, like the value of the LastPaymentDate property in the example below, or
- By defining them separately and giving them an alias, like the JohnDoesPhonNo and JohnDoesStartDate values that are first predefined and then used in the definition of the John Doe instance.

```
// ===================================================================
//  Instance of GOLF_ClubMember John Doe
// ===================================================================

instance of GOLF_Date as JohnDoesStartDate
{
    Year = 2011;
    Month = July;
    Day = 17;
};

instance of GOLF_PhoneNumber as JohnDoesPhoneNo
{
    AreaCode = {"9", "0", "7"};
    Number = {"7", "4", "7", "4", "8", "8", "4"};
};

instance of GOLF_ClubMember
{
    Caption = "Instance of John Doe\'s GOLF_ClubMember object";
    FirstName = "John";
    LastName = "Doe";
    Status = Basic;
    MembershipEstablishedDate = JohnDoesStartDate;
    MonthlyFee = 250.00;
    LastPaymentDate = instance of GOLF_Date
        {
            Year = 2011;
            Month = July;
            Day = 31;
        };
```

```
    MemberAddress = instance of GOLF_Address
        {
            State = IL;
            City = "Oak Park";
            Street "Oak Park Av.";
            StreetNo = "1177;
            AppartmentNo = "3B";
        };
    MemberPhoneNo = JohnDoesPhoneNo;
    MemberEmailAddress = "JonDoe@hotmainl.com";
};
```

The literal value, which appears on the right-hand side of the property value definitions, is defined by the following ABNF rule.

```
literalValue = integerValue / realValue / stringValue / datetimeValue / booleanValue /
               enumValue / nullValue / octetstringValue
```

The values of specific primitive types are in *Annex A*, *A.16*.

Annex A (normative)
MOF grammar description

The grammar is defined using the ABNF notation described in: http://tools.ietf.org/html/rfc5234.

The definition uses the following conventions

- All non-terminals are spelled in lower-case
- Punctuation terminals like ";" are show verbatim
- Terminal symbols are spelled in CAPITL letter when used and then defined in the keywords and symbols section (they correspond to the lexical tokens)

The grammar is written to be lexically permissive. This means that some of the MOF constraints are expected to be checked over an in-memory MOF representation specification (the ASTs) after the MOF file(s) have been parsed. For example the constraint that enumeration elements in a derived enumeration must be of the same type as the base enumeration is not encoded in the grammar. Similarly the default values of qualifier definitions are lexically permissive to keep the parsing simple.

The grammar is also written with the assumption that MOF v2 files will be converted to MOF v3 before they can be parsed by the MOF v3 compiler. For example we assume that all ValueMap and Values qualifier pairs will be translated to enumerations before a parser implementing the following grammar can be used.

A.1 MOF specification

```
mofCompilationUnit  =  *mofProduction
mofProduction       =  compilerDirective  /
                       structureDeclaration  /
                       classDeclaration   /
                       associationDeclaration  /
                       enumerationDeclaration /
                       instanceDeclaration  /
                       qualifierDeclaration
```

A.2 Compiler directive

```
compilerDirective   =  PRAGMA (pragmaName / standardPragmaName )
                       "(" pragmaParameter ")"
pragmaName          =  IDENTIFIER
standardPragmaName  =  INCLUDE
pragmaParameter     =  stringValue
PRAGMA              =  "#pragma"              ; keyword: case insensitive
INCLUDE             =  "include"              ; keyword: case insensitive
```

A.3 Structure declaration

The syntactic difference between global and nested structure declarations is that the global declarations must use schema-qualified names. This constraint can be verified after the MOF files have been parsed into the corresponding abstract syntax trees.

```
structureDeclaration  =  [ qualifierList ] [FINAL]
                         STRUCTURE structureName [superStructure]
                         "{" *propertyDeclaration "}" ";"
structureName         =  ( IDENTIFIER  / schemaQualifiedName )
superStructure        =  ":" ( IDENTIFIER / schemaQualifiedName )
FINAL                 =  “final”                ; keyword: case insensitive
STRUCTURE             =  "structure"            ; keyword: case insensitive
```

A.4 Class declaration

```
classDeclaration    =  [ qualifierList ] *2classModifier
                       [ABSTRACT] CLASS className [ superClass ]
                       "{" *classFeature "}" ";"
classModifier       =  ABSTRACT / FINAL
className           =  schemaQualifiedName
superClass          =  ":" schemaQualifiedName
classFeature        =  propertyDeclaration /
                       enumerationDeclaration /
                       structureDeclaration /
                       methodDeclaration
ABSTRACT            =  "abstract"              ; keyword: case insensitive
CLASS               =  "class"                 ; keyword: case insensitive
```

A.5 Association declaration

The only syntactic difference between the class and the association is the use of the keyword “association”.

```
associationDeclaration  =  [ qualifierList ] *2classModifier
                           ASSOCIATION associationName [ superAssociation ]
                           "{" * classFeature "}" ";"
associationName         =  schemaQualifiedName
superAssociation        =  ":" schemaQualifiedName
ASSOCIATION             =  "association"       ; keyword: case insensitive
```

A.6 Enumeration declaration

Comment [W.K.21]: Should we change the separator to “,”?

The grammar does not differentiate between derived integer and string enumerations. This is because syntactically they will be the same if literals are given no values.

```
enumDeclaration         =  [ qualifierList ] [FINAL]
                           ENUMERATION enumName ":"
                           ( integerEnumDeclaration /
                             stringEnumDeclaration /
                             derivedEnumDeclaration )
enumName                =  ( IDENTIFIER  / schemaQualifiedName )
integerEnumDeclaration  =  DT_UnsignedInteger "{" integerEnumElement *( "," integerEnumElement) "}"  ";"
stringEnumDeclaration   =  DT_STRING  "{" stringEnumElement *( "," stringEnumElement) "}"  ";"
```

```
integerEnumElement      =  [ qualifierList ] enumLiteral "=" integerValue
stringEnumElement       =  [ qualifierList ] enumLiteral [ "=" stringValue ]
derivedEnumDeclaration  =  ( IDENTIFIER  / schemaQualifiedName )
                           "{" enumElement *("," enumElement ) "}"  ";"
enumElement             =  enumLiteral [ "=" (integerValue / stringValue) ]
enumLiteral             =  IDENTIFIER
ENUMERATION             =  "enumeration"
```

Comment [W.K.22]: Modeled the values of integer enumerations as non-optional.

Comment [W.K.23]: Left string values optional.

A.7 Instance declaration

The grammar is not attempting to verify that the type of the value that is part of a property declaration is consistent with the type of the property.

Notice that by the definition of the *valueArray* an array cannot contains another array; it can only contain a collection of single-value elements.

```
instanceDeclaration     =  INSTANCE OF ( structureName / className / associationName )  [ alias ]
                           propertyValueList ";"
propertyValueList       =  "{" *propertyValue "}"
propertyValue           =  propertyName "=" value ";"
propertyName            =  IDENTIFIER
value                   =  singleValue /
                           valueArray /
singleValue             =  simpleValue /
                           complexValue
valueArray              =  "{" [ valueArrayElementList  ] "}"
valueArrayElementList   =  singleValue *("," singleValue )
simpleValue             =  literalPrimitiveTypeValue /
                           literalObjectPathValue /
                           aliasIdentifier
complexValue            =  propertyValueList /
                           instanceDeclaration
alias                   =  AS aliasIdentifier
INSTANCE                =  "instance"              ; keyword: case insensitive
AS                      =  "as"                    ; keyword: case insensitive
OF                      =  "of"                    ; keyword: case insensitive
```

Comment [W.K.24]: This is called Instance in the meta-model

Comment [W.K.25]: The value could be NULL

Comment [W.K.26]: This is questioned; should we allow “instance of X { …” to the right of the property name

The type of the value associated with a property should be consistent with the type of the property. For example the default value of a structure containing string and integer properties should be a string and an integer.

A.8 Qualifier declaration

Comment [W.K.27]: Align the syntax of the qualifier declaration with class format

Notice that qualifiers can be qualified themselves. This is to allow run-time analysis of such qualifier properties as their propagation rules.

Since many v2 qualifiers have been replaced with keywords, the MOF v2 files will have to be converted to MOF v3 before parsing.

```
qualifierDeclaration    =  [ qualifierList ] [TRANSLATABLE]
                           QUALIFIER  qualifierName ":" qualifierType qualifierScope [qualifierPolicy] ";"
qualifierName           =  IDENTIFIER
qualifierType           =  singleValueQualifierType / arrayValueQualifierType
singleValueQualifierType =  ( primitiveType / enumName ) [ "=" defaultQualifierValue ]
arrayValueQualifierType =  primitiveType "["  "]" [ "=" defaultQualifierValueList ]
defaultQualifierValue   =  literalValue
defaultQualifierValueList =  "{" primitiveTypeValueList  "}"
primitiveTypeValueList  =  literalPrimitiveTypeValue *( "," literalValue)
qualifierScope          =  "," SCOPE "(" ANY /  qualifiedElementList  ")"
qualifierPolicy         =  "," POLICY "(" APPLYMANY /  APPLYONCE / RESTRICTED ")"
qualifiedElementList    =  qualifiedElement  *(  ","  qualified Element )
qualifiedElement        =  STRUCTURE / CLASS / ASSOCIATION /
                           ENUMERATION  /  ENUMERATIONLITERAL /
                           PROPERTY / REFPROPERTY /
                           METHOD / PARAMETER /
                           QUALIFIER
TRANSLATABLE            =  "translatable"          ; keyword: case insensitive
SCOPE                   =  "scope"                 ; keyword: case insensitive
ANY                     =  "any"                   ; keyword: case insensitive
POLICY                  =  "policy"                ; keyword: case insensitive
APPLYMANY               =  "applymany"             ; keyword: case insensitive
APPLYONCE               =  "applyonce"             ; keyword: case insensitive
RESTRICTED              =   "restricted"           ; keyword: case insensitive
ENUMERATIONLITERAL      =  "enumerationliteral"    ; keyword: case insensitive
PROPERTY                =  "property"              ; keyword: case insensitive
REFPROPETY              =  "reference"             ; keyword: case insensitive
METHOD                  =  "method"                ; keyword: case insensitive
PARAMETER               =  "parameter"             ; keyword: case insensitive
QUALIFIER               =  "qualifier"             ; keyword: case insensitive
```

Comment [W.K.28]: What is the default value if the policy is not specified? ApplyMany?

A.9 Qualifier list

```
qualifierList              = "[" qualifier *( "," qualifier ) "]"
qualifier                  = qualifierName [ qualifierParameter ]
qualifierName              = IDENTIFIER
qualifierParameter         = qualifierValueInitializer / qualiferValueArrayInitializer
qualifierValueInitializer  = "(" literalPrimitiveTypeValue ")"
qualiferValueArrayInitializer = "{" primitiveTypeValue *( "," primitiveTypeValue ) "}"
```

A.10 Property declaration

```
propertyDeclaration        = [ qualifierList ]  [KEY]
                             ( basePropertyDeclaration /
                               structurePropertyDeclaration /
                               enumPropertyDeclaration /
                               classPropertyDeclaration /
                               refereceePropertyDeclaration )
basePropertyDeclaration    = primitiveType propertyName [ array ] [ "=" primitiveTypeDefaultValue ] ";"
structurePropertyDeclaration = strucutreName propertyName [ array ] [  "=" structureDefaultValue ] ";"
enumPropertyDeclaration    = enumName propertyName [ array ] [  "=" enumDefaultValue ] ";"
classPropertyDeclaration   = className propertyName [ array ] ["=" structureDefaultValue ] ";"
refereceePropertyDeclaration = className REF propertyName [ array ] [ "=" referenceDefaultValue ] ";"
array                      = "[" "]"
propertyName               = IDENTIFIER
KEY                        = "key"                    ; keyword: case insensitive
REF                        = "ref"                    ; keyword: case insensitive
```

Comment [W.K.29]: Array size constraints should be specified using the Constraint qualifier => OCL.

A.11 Method declaration

```
methodDeclaration          = [ qualifierList ] [ STATIC ] ( dataType [ array ] / VOID )  methodName
                             "(" [parameterList ] ")" ";"
methodName                 = IDENTIFIER
VOID                       = "void"                   ; keyword: case insensitive
STATIC                     = “static”                 ; keyword: case insensitive
parameterList              = parameterDeclaration *( "," parameterDeclaration )
```

A.12 Parameter declaration

```
parameterDeclaration = [ qualifierList ] [ IN / OUT / INOUT ]
                       ( primitiveParamDeclaration /
                         structureParamDeclaration /
                         enumParamDeclararion /
                         classParamDeclararion /
                         referenceParamDeclaration )
primitiveParamDeclaration = primitiveType parameterName [ array ] [ "="  primitiveTypeDefaultValue ]
structureParamDeclartion = structureName parameterName [ array ] ["=" structureDefaultValue ]
enumParamDeclaration = enumName parameterName [ array ]    ["=" enumDefaultValue ]
classParamDeclaration = className parameterName [ array ] ["="structureDefaultValue]
referenceParamDeclaration = className "REF" parameterName [ array ] [ "=" referenceDefaultValue ]
parameterName = IDENTIFIER
IN = "in"                    ; keyword: case insensitive
OUT = "out"                  ; keyword: case insensitive
INOUT = "inout"              ; keyword: case insensitive
```

A.13 Values

```
primitiveTypeDefaultValue = literalPrimitiveTypeValue / literalValueArray
literalValueArray = "{" literalPrimitiveTypeValue *( "," literalPrimitiveTypeValue ) "}"
literalValue = integerValue / realValue / stringValue / datetimeValue / booleanValue /
               enumValue / nullValue / octetstringValue
integerValue = binaryValue / octalSignedValue / decimalValue / hexValue
octetstringValue = octalValue
enumValue = [enumName"."] enumLiteral
structureDefaultValue = complexValue / complexValueArray
complexValueArray = "{" complexValue *( "," complexValue) "}"
enumDefaultValue = stringValue
referenceDefaultValue = referenceValue / referenceValueArray
referenceValue = literalObjectPathValue / aliasIdentifier
referenceValueArray = "{" referenceValue *( "," referenceValue ) "}"
literalObjectPathValue = stringValue
```

Comment [W.K.30]: Should we define the structure of the object path string in MOF?

A.14 Base data types

```
primitiveType          = DT_Integer /
                         DT_REAL32 /
                         DT_REAL64 /
                         DT_STRING /
                         DT_DATETIME /
                         DT_BOOLEAN /
                         DT_OCTETSTRING
DT_Integer             = DT_UnsignedInteger /
                         DT_SignedInteger
DT_UnsignedInteger     = DT_UINT8 /
                         DT_UINT16 /
                         DT_UINT32 /
                         DT_UINT64
                         DT_UINT128
DT_SignedInteger       = DT_SINT8 /
                         DT_SINT16 /
                         DT_SINT32 /
                         DT_SINT64 /
                         DT_SINT128 /
DT_UINT8               = "uint8"          ; keyword: case insensitive
DT_UINT16              = "uint16"         ; keyword: case insensitive
DT_UINT32              = "uint32"         ; keyword: case insensitive
DT_UINT64              = "uint64"         ; keyword: case insensitive
DT_UINT128             = "uint128"        ; keyword: case insensitive
DT_SINT8               = "sint8"          ; keyword: case insensitive
DT_SINT16              = "sint16"         ; keyword: case insensitive
DT_SINT32              = "sint32"         ; keyword: case insensitive
DT_SINT64              = "sint64"         ; keyword: case insensitive
DT_SINT128             = "sint128"        ; keyword: case insensitive
DT_REAL32              = "real32"         ; keyword: case insensitive
DT_REAL64              = "real64"         ; keyword: case insensitive
```

```
DT_DATETIME     = "datetime"      ; keyword: case insensitive
DT_STRING       = “string”        ; keyword: case insensitive
DT_BOOLEAN      = “boolean”       ; keyword: case insensitive
DT_OCTETSTRING  = “octetstring”   ; keyword: case insensitive
```

A.15 Names

The values recognized by the following are case independent. They also don’t allow whitespace.

```
schemaQualifiedName = schemaName UNDERSCORE IDENTIFIER
schemaName          = firstSchemaChar *( nextSchemaChar )
firstSchemaChar     = UPPERALPHA / LOWERALPHA
nextSchemaChar      = firstSchemaChar / decimalDigit

IDENTIFIER          = firstIdentifierChar *( nextIdentifierChar )
firstIdentifierChar = UPPERALPHA / LOWERALPHA / UNDERSCORE
nextIdentifierChar  = firstIdentifierChar / decimalDigit
aliasIdentifier     = "$" IDENTIFIER
```

A.16 Primitive type values

```
; Whitespace is allowed between the double-quoted parts.
; The combined date-time string value shall conform to the format defined by the dt-format ABNF rule,
; with further constraints on the field values as defined in DSP0004.
datetimeValue       = DOUBLEQUOTE dt-format DOUBLEQUOTE
dt-format           = dt-timestampValue / dt-intervalValue

dt-timestampValue   = 14*14(decimalDigit) "." dt-microseconds ( "+" / "-" ) dt-timezone
                    / dt-yyyymmddhhmmss "." 6*6( "*" ) ( "+" / "-" ) dt-timezone
dt-yyyymmddhhmmss   = 12*12(decimalDigit) 2*2( "*" )
                    / 10*10(decimalDigit) 4*4( "*" )
                    / 8*8(decimalDigit) 6*6( "*" )
                    / 6*6(decimalDigit) 8*8( "*" )
                    / 4*4(decimalDigit) 10*10( "*" )
                    / 14*14( "*" )
```

```
dt-timezone                 = 3*3(decimalDigit)

dt-intervalValue            = 14*14(decimalDigit) "." dt-microseconds ":" "000" /
                              dt-ddddddddhhmmss "." 6*6("*") ":" "000"
dt-ddddddddhhmmss           = 12*12(decimalDigit) 2*2("*")
                               / 10*10(decimalDigit) 4*4("*")
                              / 8*8(decimalDigit) 6*6("*")
                               / 14*14("*")
dt-microseconds             = 6*6(decimalDigit)
                              / 5*5(decimalDigit) 1*1("*")
                              / 4*4(decimalDigit) 2*2("*")
                              / 3*3(decimalDigit) 3*3("*")
                              / 2*2(decimalDigit) 4*4("*")
                              / 1*1(decimalDigit) 5*5("*")
                              / 6*6("*")

; Whitespace and comment is allowed between double quoted parts.
; Double quotes shall be escaped. Tthe (unescaped) contents of stringValue shall conform to the
; string representation for object paths.
stringValue                 = 1*( DOUBLEQUOTE *stringChar DOUBLEQUOTE)
stringChar                  = UCScharString / stringEscapeSequence
UCScharString               = 1*6(hexDigit)             ; any UCS character for use in string constants.
stringEscapeSequence        = BACKSLASH ( BACKSPACE / TAB / LINEFEED / FORMFEED / RETURN /
                              DOUBLEQUOTE / SINGLEQUOTE / BACKSLASH / UCSchar )
BACKSPACE                   = %x08                      ; U+0008: backspace
TAB                         = %x09                       ; U+0009: horizontal tab
LINEFEED                    = %x0A                       ; U+000A: linefeed
FORMFEED                    = %x0C                       ; U+000C: form feed
RETURN                      = %x0D                       ; U+000D: carriage return
DOUBLEQUOTE                 = %x22                      ; U+0022: double quote (")
SINGLEQUOTE                 = %x27                       ; U+0027: single quote (')
BACKSLASH                   = %x5C                       ; U+005C: backslash (\)
UPPERALPHA                  = %x41-5A                   ; U+0041-U+005A or "A" ... "Z"
LOWERALPHA                  = %x61-7A                   ; U+0061-U+007A or "a" ... "z"
```

```
UNDERSCORE          = %x5F                  ; U+005F or "_"
UCSchar             = ( "x" / "X" ) 1*4( hexDigit ) ; no whitespace: UCS code position

; The following ABNF rules do not allow whitespace, unless stated otherwise.
booleanValue        = TRUE / FALSE
FALSE               = "false"               ; keyword: case Insensitive
TRUE                = "true"                ; keyword: case Insensitive

; The following ABNF rules do not allow whitespace, unless stated otherwise:
nullValue           = NULL
NULL                = "null"                ; keyword: case Insensitive

; The following ABNF rules do not allow whitespace, unless stated otherwise.
binaryValue         = [ "+" / "-" ] 1*binaryDigit ( "b" / "B" )
binaryDigit         = "0" / "1"
octalValue          = "0" 1*octalDigit
octalSignedValue    = [ "+" / "-" ] octalValue
octalDigit          = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7"
hexValue            = [ "+" / "-" ] ( "0x" / "0X" ) 1*hexDigit
hexDigit            = decimalDigit / "a" / "A" / "b" / "B" / "c" / "C" /
                      "d" / "D" / "e" / "E" / "f" / "F"
decimalValue        = [ "+" / "-" ] ( positiveDecimalDigit *decimalDigit / "0" )
realValue           = [ "+" / "-" ] *decimalDigit "." 1*decimalDigit
                      [ ( "e" / "E" ) [ "+" / "-" ] 1*decimalDigit ]
decimalDigit        = "0" / positiveDecimalDigit
positiveDecimalDigit = "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"
```

Annex B (normative) MOF keywords

Below are the MOF v3 keyword listed in alphabetical order.

#pragma	true
abstract	scope
any	sint8
applymany	sint16
applyonce	sint32
as	sint64
association	sint128
boolean	static
class	string
datetime	structure
enumeration	translatable
enumerationliteral	uint8
false	uint16
in	uint32
include	uint64
inout	uint128
key	void
method	qualifier
null	
octetstring	
of	
out	
parameter	
policy	
property	
real32	
real64	
restricted	
ref	
reference	

Annex C (informative)
Example MOF specitication

The following is the content of the MOF files in the example GOLF model specification.

C.1 GOLF_Schema.mof

```
//
//     Copyright 2011 Distributed Management Task Force, Inc. (DMTF).
//     Exampel domain used to illustrate CIM v3 and MOF v3 features
//
#pragma include ("GOLF_Base.mof")
#pragma include ("GOLF_Club.mof")
#pragma include ("GOLF_ClubMember.mof")
#pragma include ("GOLF_Professional.mof")
#pragma include ("GOLF_Locker.mof")
#pragma include ("GOLF_MemberLocker.mof")
#pragma include ("GOLF_Lesson.mof")
//
//     Global structures
//
#pragma include ("GlobalStructs/GOLF_Address.mof")
#pragma include ("GlobalStructs/GOLF_Date.mof")
#pragma include ("GlobalStructs/GOLF_PhoneNumber.mof")
//
//  Global enumerations
//
#pragma include ("GlobalEnums/GOLF_ResultCodeEnum.mof")
#pragma include ("GlobalEnums/GOLF_GOLF_MonthsEnum.mof")
#pragma include ("GlobalEnums/GOLF_GOLF_StatesEnum.mof")
//
//  Instances
//
#pragma include ("Instances/JohnDoe.mof")
```

C.2 GOLF_Base.mof

```
// ===================================================================
//  GOLF_Base
// ===================================================================
abstract class GOLF_Base {
     [Description (
        "InstanceID is a property that opaquely and uniquely identifies "
             "an instance of a class that derives from the GOLF_Base class. ")]
   key string InstanceID;

     [Description (
```

```
        "A a short textual description (one- line string) of the "
            "instance." ),
      MaxLen (64)]
  string Caption;
};
```

C.3 GOLF_Club.mof

```
// ===================================================================
//  GOLF_Club
// ===================================================================
  [Description (
          "Instances of this class represent golf clubs. A golf club is an "
          "an organization that provides member  services to golf players "
          "bouth amateur and professional." )]
class GOLF_Club: GOLF_Base {
// ============================ properties ============================
      string ClubName;
      GOLF_Date YearEstablished;

      GOLF_Address ClubAddress;
      GOLF_Phone ClubPhoneNo;
      GOLF_Fax ClubFaxNo;
      string ClubWebSiteURL;

      GOLF_ClubMember REF AllMembers[];
      GOLF_Professional REF Professionals[];

// ============================= methods ==============================
      static GOLF_ResultCodeEnum AddNonProfessionalMember (
          in GOLF_ClubMember newMember
      );

      static GOLF_ResultCodeEnum AddProfessional (
          in GOLF_Professional newProfessional
      );

      static UInt32 GetMembersWithOutstandingFees (
          in GOLF_Date referenceDate,
          out GOLF_ClubMember REF lateMembers[]
      );

      static GOLF_ResultCodeEnum TerminateMembership (
          in GOLF_ClubMember REF member
      );
};
```

C.4 GOLF_ClubMember.mof

```
// ===================================================================
```

```
// GOLF_ClubMember
// ===================================================================
  [Description (
        "Instances of this class represent members of a golf club." )]
class GOLF_ClubMember: GOLF_Base {

// ====================== embeded enumerations ======================
    enumeration MemberStatusEnum : Uint16 {
        Basic = 0,
        Extended = 1,
        VP = 2
    };

// ============================ properties ============================
    string FirstName;
    string LastName;
    MemberStatusEnum Status;
    GOLF_Date MembershipEstablishedDate;

    Real32 MembershipSignUpFee;
    Real32 MonthlyFee;
    GOLF_Date LastPaymentDate;

    GOLF_Address MemberAddress;
        [ Constraint ("inv: not oclIsUndefined()") ]
    GOLF_PhoneNumber MemberPhoneNo;
    string MemberEmailAddress;

// ============================= methods ==============================
    GOLF_ResultCodeEnum SendPaymentReminderMessage();
};
```

C.5 GOLF_Professional.mof

```
// ===================================================================
// GOLF_Professional
// ===================================================================
class GOLF_Professional : GOLF_ClubMember {
// ====================== embeded structures ======================
    structure Sponsor {
        string Name,
        GOLF_Date ContractSignedDate;
        Real32 ContractAmount;
    };

// ====================== embeded enumerations ======================
    enumeration ProfessionalStatusEnum : MemberStatusEnum {
        Professional = 6,
        SponsoredProfessional = 7
```

```
    };

// ============================ properties ============================
        [Override]
    ProfessionalStatusEnum Status = Professional;
    Sponsor Sponsors[];
    Boolean Ranked;

// ============================= methods ==============================
    static GOLF_ResultCodeEnum GetNumberOfProfessionals (
        out Uint32 NoOfPros,
        in ProfessionalStatusEnum = Professional
    )
};
```

C.6 GOLF_Locker.mof

```
// ===================================================================
//  GOLF_Locker
// ===================================================================
class GOLF_Locker : GOLF_Base {
    string Location;
    Uint16 LockerNo;
    Real32 MonthlyRentFee;
};
```

C.7 GOLF_MemberLocker.mof

```
// ===================================================================
//  GOLF_MemberLocker
// ===================================================================
association GOLF_MemberLocker : GOLF_Base {
        [Max(1)]
    GOLF_Member REF Member;
        [Max(1)]
    GOLF_Locker REF Locker;
    GOLF_Date AssignedOnDate;
};
```

C.8 GOLF_Lesson.mof

```
// ===================================================================
//  GOLF_Lesson
// ===================================================================
association GOLF_Lesson : GOLF_Base {
    GOLF_Professional REF Instructur;
    GOLF_ClubMember REF Student;

    DateTime Schedule;
        [Description ( "The duration of the lesson in minutes" )]
```

```
    UInt16 Duration;
    String Location;
    Real32 LessonFee;
};
```

C.9 GOLF_Address.mof

```
// ===================================================================
//  GOLF_Address
// ===================================================================
structure GOLF_Address {
    GOLF_StateEnum State;
    string City;
    string Street;
    string StreetNo;
    string AppartmentNo;
};
```

C.10 GOLF_Date.mof

```
// ===================================================================
//  GOLF_Date
// ===================================================================
structure GOLF_Date {
    UInt16 Year = 2000;
    GOLF_MonthsEnum Month = GOLF_MonthsEnum.January;
        [MinValue(1), MaxValue(31)]
    UInt16 Day = 1;
};
```

C.11 GOLF_PhoneNumber.mof

```
// ===================================================================
//  GOLF_PhoneNumber
// ===================================================================
structure GOLF_PhoneNumber {
        [Constraint ("inv: size() = 3")]
    Char16 AreaCode[];
        [Constraint ("inv: size() = 7")]
    Char16 Number[];
};
```

C.12 GOLF_ResultCodeEnum.mof

```
// ===================================================================
//  GOLF_ResultCodeEnum
// ===================================================================
enumeration GOLF_ResultCodeEnum : UInt32 {
   // The operation was successful
   RESULT_OK = 0,
   // A general error occurred, not covered by a more specific error code.
   RESULT_FAILED = 1,
   // Access to a CIM resource is not available to the client.
```

```
    RESULT_ACCESS_DENIED = 2,
    // The target namespace does not exist.
    RESULT_INVALID_NAMESPACE = 3,
    // One or more parameter values passed to the method are not valid.
    RESULT_INVALID_PARAMETER  = 4,
    // The specified class does not exist.
    RESULT_INVALID_CLASS = 5,
    // The requested object cannot be found.
    RESULT_NOT_FOUND = 6,
    // The requested operation is not supported.
    RESULT_NOT_SUPPORTED = 7,
    // The operation cannot be invoked because the class has subclasses.
    RESULT_CLASS_HAS_CHILDREN = 8,
    // The operation cannot be invoked because the class has instances.
    RESULT_CLASS_HAS_INSTANCES = 9,
    // The operation cannot be invoked because the superclass does not exist.
    RESULT_INVALID_SUPERCLASS = 10,
    // The operation cannot be invoked because an object already exists.
    RESULT_ALREADY_EXISTS = 11,
    // The specified property does not exist.
    RESULT_NO_SUCH_PROPERTY = 12,
    // The value supplied is not compatible with the type.
    RESULT_TYPE_MISMATCH = 13,
    // The query language is not recognized or supported.
    RESULT_QUERY_LANGUAGE_NOT_SUPPORTED = 14,
    // The query is not valid for the specified query language.
    RESULT_INVALID_QUERY = 15,
    // The extrinsic method cannot be invoked.
    RESULT_METHOD_NOT_AVAILABLE = 16,
    // The specified extrinsic method does not exist.
    RESULT_METHOD_NOT_FOUND = 17,
    // The specified namespace is not empty.
    RESULT_NAMESPACE_NOT_EMPTY = 20,
    // The enumeration identified by the specified context is invalid.
    RESULT_INVALID_ENUMERATION_CONTEXT = 21,
    // The specified operation timeout is not supported by the CIM Server.
    RESULT_INVALID_OPERATION_TIMEOUT = 22,
    // The Pull operation has been abandoned.
    RESULT_PULL_HAS_BEEN_ABANDONED = 23,
    // The attempt to abandon a concurrent Pull operation failed.
    RESULT_PULL_CANNOT_BE_ABANDONED = 24,
    // Using a filter in the enumeration is not supported by the CIM server.
    RESULT_FILTERED_ENUMERATION_NOT_SUPPORTED = 25,
    // The CIM server does not support continuation on error.
    RESULT_CONTINUATION_ON_ERROR_NOT_SUPPORTED = 26,
    // The operation failed because server limits were exceeded.
    RESULT_SERVER_LIMITS_EXCEEDED = 27,
    // The CIM server is shutting down and cannot process the operation.
```

```
    RESULT_SERVER_IS_SHUTTING_DOWN = 28
};
```

C.13 GOLF_MonthsEnum.enum

```
// ==================================================================
//  GOLF_MonthsEnum
// ==================================================================
enumeration GOLF_MonthsEnum : String {
      January,
      February,
      March,
      April,
      May,
      June,
      July,
      August,
      September,
      October,
      November,
      December
};
```

C.14 GOLF_StatesEnum.mof

```
// ==================================================================
//  GOLF_StatesEnum
// ==================================================================
enumeration GOLF_StatesEnum : string {
      AL = "Alabama",
      AK = "Alaska",
      AZ = "Arizona",
      AR = "Arkansas",
      CA = "California",
      CO = "Colorado",
      CT = "Connecticut",
      DE = "Delaware",
      FL = "Florida",
      GA = "Georgia",
      HI = "Hawaii",
      ID = "Idaho",
      IL = "Illinois",
      IN = "Indiana",
      IA = "Iowa",
      KS = "Kansas",
      LA = "Louisiana",
      ME = "Main",
      MD = "Maryland",
      MA = "Massachusetts",
      MI = "Michigan",
```

```
    MS = "Mississippi",
    MO = "Missouri",
    MT = "Montana",
    NE = "Nebraska",
    NV = "Nevada",
    NH = "New Hampshire",
    NJ = "New Jersey",
    NM = "New Mexico",
    NY = "New York",
    NC = "North Carolina",
    ND = "North Dakota",
    OH = "Ohio",
    OK = "Oklahoma",
    OR = "Oregon",
    PA = "Pennsylvania",
    RI = "Rhode Island",
    SC = "South Carolina",
    SD = "South Dakota",
    TX = "Texas",
    UT = "Utah",
    VT = "Vermont",
    VA = "Virginia",
    WA = "Washington",
    WV = "West Virginia",
    WI = "Wisconsin",
    WY = "Wyoming"
};
```

C.15 JohnDoe.mof

```
// ===================================================================
//  Instance of GOLF_ClubMember John Doe
// ===================================================================

instance of GOLF_Date as JohnDoesStartDate
{
    Year = 2011;
    Month = July;
    Day = 17;
};

instance of GOLF_PhoneNumber as JohnDoesPhoneNo
{
    AreaCode = {"9", "0", "7"};
    Number = {"7", "4", "7", "4", "8", "8", "4"};
};

instance of GOLF_ClubMember
{
```

```
    Caption = "Instance of John Doe\'s GOLF_ClubMember object";
    FirstName = "John";
    LastName = "Doe";
    Status = Basic;
    MembershipEstablishedDate = JohnDoesStartDate;
    MonthlyFee = 250.00;
    LastPaymentDate = instance of GOLF_Date
        {
            Year = 2011;
            Month = July;
            Day = 31;
        };
    MemberAddress = instance of GOLF_Address
        {
            State = IL;
            City = "Oak Park";
            Street "Oak Park Av.";
            StreetNo = "1177;
            AppartmentNo = "3B";
        };
    MemberPhoneNo = JohnDoesPhoneNo;
    MemberEmailAddress = "JonDoe@hotmainl.com";
};
```