Document Number: DSP0221

Date: 2012-12-13

Version: 3.0.0

Managed Object Format (MOF)

Document Type: Specification

Document Status: DMTF Standard

Document Language: en-US

DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability. Members and non-members may reproduce DMTF specifications and documents, provided that correct attribution is given. As DMTF specifications may be revised from time to time, the particular version and release date should always be noted.

Implementation of certain elements of this standard or proposed standard may be subject to third party patent rights, including provisional patent rights (herein "patent rights"). DMTF makes no representations to users of the standard as to the existence of such rights, and is not responsible to recognize, disclose, or identify any or all such third party patent right, owners or claimants, nor for any incomplete or inaccurate identification or disclosure of such rights, owners or claimants. DMTF shall have no liability to any party, in any manner or circumstance, under any legal theory whatsoever, for failure to recognize, disclose, or identify any such third party patent rights, or for such party's reliance on the standard or incorporation thereof in its product, protocols or testing procedures. DMTF shall have no liability to any party implementing such standard, whether such implementation is foreseeable or not, nor to any patent owner or claimant, and shall have no liability or responsibility for costs or losses incurred if a standard is withdrawn or modified after publication, and shall be indemnified and held harmless by any party implementing the standard from any and all claims of infringement by a patent owner for such implementations.

For information about patents held by third-parties which have notified the DMTF that, in their opinion, such patent may relate to or impact implementations of DMTF standards, visit http://www.dmtf.org/about/policies/disclosures.php.

Contents

Figures

Tables

Foreword

The *Managed Object Format (MOF)* specification (this document) 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:

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

Introduction

This document specifies the DMTF *Managed Object Format (MOF),* which is a schema description language used for specifying the interface of managed resources (storage, networking, compute, software) conformant with the CIM Metamodel defined in DSP0004.

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.

Deprecated material

Deprecated material is not recommended for use in new development efforts. Existing and new implementations may use this material, but they should 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 can use deprecated elements, they are directed to use the favored elements instead.

Deprecated material should contain references to the last published version that included it as normative, 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 included in this document is an aid to implementers who are interested in likely future developments. Experimental material might change as implementation experience is gained. Until included in future documents as normative, all 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.

Managed Object Format (MOF)

1 Scope

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

The MOF provides the means to write interface definitions of managed resource types including their properties, behavior and relationships with other objects. Instances of managed resource types represent logical concepts like policies, as well as real-world resource such as disk drives, network routers or software components.

MOF is used to define industry-standard managed resource types, published by the DMTF as the CIM Schema and other schemas, as well as user/vendor-defined resource 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 cited edition (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) Metamodel 3.0*
http://www.dmtf.org/sites/default/files/standards/documents/DSP0004_3.0.pdf

IETF RFC3986, *Unified Resource Identifier (URI): General Syntax, January 2005*
http://tools.ietf.org/html/rfc3986

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

ISO/IEC 80000-13:2008, *Quantities and units, Part13*
http://www.iso.org/iso/catalogue_detail.htm?csnumber=31898

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/IEC 10646:2012, *Information technology -- Universal Coded Character Set (UCS)*
http://standards.iso.org/ittf/PubliclyAvailableStandards/c056921_ISO_IEC_10646_2012.zip

OMG, *Object Constraint Language, Version 2.3.1*
http://www.omg.org/spec/OCL/2.3.1

The Unicode Consortium, Unicode 6.1.0, *Unicode Standard Annex #15: Unicode Normalization Forms*
http://www.unicode.org/reports/tr15/tr15-35.html

3 Terms and definitions

Some terms used in this document have a specific meaning beyond the common English interpretation. 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 defined in DSP0004 apply to this document. The following additional terms are used in this document.

3.1

Managed Object Format

Refers to the language described in this specification.

3.2

MOF grammar

Refers to the MOF language syntax description included in this document. The MOF grammar is specified using the ABNF (see RFC5234).

3.3

MOF file

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

3.4

MOF compilation unit

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

3.5

MOF compiler

A MOF compiler takes as input a compilation unit, and in addition can also accept as input a representation of previously compiled types and qualifiers.

A MOF compiler transforms types defined in the compilation unit into another representation, like schema repository entries or provider skeletons.

A MOF compiler shall verify the consistency of its input; the compiler input shall include definitions of all types that are used by other types, and all super-types of the defined and used types.

4 Symbols and abbreviated terms

The abbreviations defined in DSP0004 apply to this document. The following additional abbreviations are used in this document.

4.1

AST
Abstract Syntax Tree

4.2
MOF
Managed Object Format

4.3
ABNF
Augmented BNF (see RFC5234)

4.4
IDL
Interface Definition Language (see ISO/IEC 14750)

4.5
OCL
Object Constraint Language (see OMG Object Constraint 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 (Unicode, Annex 15) and in the coded representation form UTF-8 (ISO 10646).

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

5.2 Whitespace

Whitespace in a MOF file is any combination of the following characters:

- Space (U+0020),
- Horizontal Tab (U+0009),
- Carriage Return (U+000D) and
- Line Feed (U+000A).

The `WS` ABNF rule represents any one of these whitespace characters:

```
WS = U+0020 / U+0009 / U+000D / U+000A
```

5.3 Line termination

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

- A Carriage Return (U+000D) followed by Line Feed (U+000A)
- A Carriage Return (U+000D) not followed by Line Feed (U+000A)
- A Line Feed (U+000A) not preceded by a Carriage Return (U+000D)

- 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 if they were whitespace.

Comments may appear anywhere in MOF syntax where whitespace is allowed and are indicated by either a leading double slash (`//`) or a pair of matching `/*` and `*/` character sequences. Occurrences of these character sequences in string literals shall not be treated as comments.

A `//` comment is terminated by the end of line (see 5.3), as shown in the example below.

```
uint16 MyProperty;  // This is an example of a single-line comment
```

A comment that begins with `/*` is terminated by the next `*/` sequence, or by the end of the MOF file, whichever comes first.

```
/* example of a comment between property definition tokens and a multi-line comment */
uint16 /* 16-bit integer property  */  MyProperty;  /* and a multi-line
                        comment */
```

6 MOF and OCL

This MOF language specification refers to OCL in two contexts:

- It refers to specific OCL constraints of the CIM Metamodel, which are defined in DSP0004.
- A schema specified in MOF may include zero or more OCL qualifiers, where each of those qualifiers contains at least one OCL statement. The statements on a qualifier should be interpreted as a collection. For example a variable defined in one statement can be used in another statement.

The OCL rules defined in CIM Metamodel specify the schema integrity rules that a MOF compiler shall check. For example one of those rules states that a structure cannot inherit from another structure that has been qualified as terminal, and therefore MOF compliers shall implement a corresponding model integrity validation rule. The CIM Metamodel constraints are specified in clause 6 of DSP0004 and then listed in ANNEX G of that document.

Within a user-defined schema, an OCL qualifier is used to define rules that all instances of the qualified element shall conform to. As an example, consider a class-level OCL qualifier that defines an invariant, which states that one of the class properties must be always greater than another of its properties. The implementations of the schema should assure that all instances of that class satisfy that condition. This has the following implications for the MOF compiler developers and the provider developers:

- The MOF compilers should parse the content of the OCL qualifiers and verify
  - conformance of the OCL expressions with the OCL syntax defined in the OMG Object Constraint Language
  - consistency of the statements with the schema elements
- The provider developers should implement the logic, which assures that resource instances conform to the requirements specified by the schema, including those specified as the OCL constraints.

7 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 compilation unit
- Qualifier declarations
- Type declarations such as classes, structures or enumerations
- Instance and value specifications

Elements of MOF language are introduced and exemplified one at a time, in a sequence that progressively builds a meaningful MOF specification. To make the examples consistent, the document uses a small, fictitious, and simplified golf club membership schema. The files of the schema are listed in ANNEX E.

A complete description of the MOF grammar is in ANNEX A.

7.1 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 format defined by ABNF rule `compilerDirective` (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

```
compilerDirective      = PRAGMA directiveName "(" stringValue ")"
PRAGMA                 = "#pragma"            ; keyword: case insensitive
directiveName          = IDENTIFIER
```

where `IDENTIFIER` is defined in A.13.

The current standard compiler directives are listed in Table 1.

Table 1 – Standard compiler directives

Compiler Directive	Description
#pragma include (<filePath>)	The included directive specifies that the referenced MOF specification file should be included in the compilation unit. The content of the referenced file shall be textually inserted in place of the directive. The included file name 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 the pragma. The format of <filePath> is defined in A.17.8.

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 include the underscore (_, U+005F). Therefore, it is recommended that the names of vendor-specific compiler directives conform to the following format (no whitespace is allowed between the elements of this ABNF rule):

```
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 and should be ignored. Thus, the use of vendor-specific compiler directives may affect the interoperability of MOF.

7.2 Qualifiers

A qualifier is a named and typed metadata element associated with a schema element, such as a class or method, and it provides information about or specifies the behavior of the qualified element. A detailed discussion of the qualifier concept is in subclause 5.6.12 of DSP0004, and the list of standard qualifiers is in clause 7 of DSP0004.

Each qualifier is defined by its qualifier type declaration. The `qualifierTypeDeclaration` MOF grammar rule corresponds to the QualifierType CIM Metamodel element defined in DSP0004, and is defined by the following ABNF rules (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

```
qualifierTypeDeclaration = [ qualifierList ] QUALIFIER  qualifierName ":"
                          qualifierType qualifierScope
                          [ qualifierPolicy ] ";"
qualifierName           = elementName
qualifierType           = primitiveQualifierType / enumQualiferType
primitiveQualifierType  = primitiveType [ array ]
                          [ "=" primitiveTypeValue ] ";"
enumQualiferType        = enumName [ array ]  "=" enumTypeValue ";"
qualifierScope          = SCOPE "(" ANY / scopeKindList ")"
qualifierPolicy         = POLICY "(" policyKind ")"
policyKind              = DISABLEOVERRIDE /
                          ENABLEOVERRIDE /
                          RESTRICTED
scopeKindList           = scopeKind  *( "," scopeKind )
scopeKind               = STRUCTURE / CLASS / ASSOCIATION /
                          ENUMERATION / ENUMERATIONVALUE /
                          PROPERTY / REFPROPERTY /
                          METHOD / PARAMETER /
                          QUALIFIERTYPE
SCOPE                   = "scope"               ; keyword: case insensitive
ANY                     = "any"                 ; keyword: case insensitive
POLICY                  = "policy"              ; keyword: case insensitive
ENABLEOVERRIDE          = "enableoverride"      ; keyword: case insensitive
```

```
DISABLEOVERRIDE          = "disableoverride"     ; keyword: case insensitive
RESTRICTED               = "restricted"          ; keyword: case insensitive
ENUMERATIONVALUE         = "enumerationvalue"    ; keyword: case insensitive
PROPERTY                 = "property"            ; keyword: case insensitive
REFPROPERTY              = "reference"           ; keyword: case insensitive
METHOD                   = "method"              ; keyword: case insensitive
PARAMETER                = "parameter"           ; keyword: case insensitive
QUALIFIERTYPE            = "qualifiertype"       ; keyword: case insensitive
```

Only numeric and Boolean primitive qualifier types (see `primitiveQualifierType` above) can be specified without specifying a value. If not specified, the implied value is as follows:

- For data type Boolean, the implied value is True.
- For numeric data types, the implied value is Null.
- For arrays of numeric or Boolean data type, the implied value is that the array is empty.

For all other types, including enumeration qualifier types (see `enumQualiferType` above), the value must be defined.

The following MOF fragment is an example of the qualifier type AggregationKind. The AggregationKind qualifier type defines the enumeration values that are used on properties of associations that are references, to indicate the kind of aggregation they represent. The type of the qualifier is an enumeration with three values; None, Shared, and Exclusive.

```
[Description ("The value of this qualifier indicates the kind of aggregation "
   "relationship defined between instances of the class containing the qualified "
   "reference property and instances referenced by that property. The value may "
   "indicate that the kind of aggregation is unspecified.")]
Qualifier AggregationKind : CIM_AggregationKindEnum = None
    Scope(reference) Flavor (disableoverride);

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

The `qualifierValue` rule in MOF corresponds to the Qualifier CIM Metamodel element defined in DSP0004, and defines the representation of an instance of a qualifier. A list of qualifier values describing a schema element shall conform to the following `qualifierList` ABNF rule (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

```
qualifierList            = "[" qualifierValue *( "," qualifierValue ) "]"
qualifierValue           = qualifierName [ qualifierValueInitializer /
                           qualiferValueArrayInitializer ]
qualifierValueInitializer     = "(" literalValue ")"
qualiferValueArrayInitializer = "{" literalValue *( "," literalValue ) "}"
```

The list of qualifier scopes (see the `scopeKind` rule above) includes "qualifiertype", which implies that qualifier declarations can be themselves qualified. Examples of standard qualifiers that can be used to describe a qualifier declaration are Description and Deprecated.

7.3 Types

CIM Metamodel defines the following hierarchy of types:

- Structure
  - Class
    - Association
- Enumeration
- Primitive type, and
- Reference type.

CIM Metamodel has a predefined list of primitive types, and their MOF representations are described in 7.3.5 and in A.15.

Elements of type reference represent references to instances of class. The declarations of properties and method parameters of type reference are described in subclauses 7.3.2 and 7.3.3, respectively. The representation of the reference type value is described in A.18.

Structures, classes, associations, and enumerations are types defined in a schema. The following sub-clauses describe how those types are declared using MOF.

7.3.1 Enumeration declaration

There are two kinds of enumerations in CIM:

- Integer enumerations
- String enumerations

Integer enumerations, which are comparable to enumerations in programming languages, represent enumeration values as distinct integer values.

String enumerations, which can be found in UML and are similar to XML enumerations (see XML Schema, Part2: Datatypes), represent enumeration values as distinct string values that in most cases are identical to the values themselves.

The `enumDeclaration` MOF grammar rule corresponds to the Enumeration CIM Metamodel element defined in DSP0004, and conforms to the following ABNF rules (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

```
enumDeclaration          = enumTypeHeader enumName ":" enumTypeDeclaration ";"
enumTypeHeader           = [ qualifierList ] ENUMERATION
enumName                 = elementName
enumTypeDeclaration      = (DT_Integer / integerEnumName )
                           integerEnumDeclaration /
                           (DT_STRING / stringEnumName) stringEnumDeclaration
integerEnumName          = enumName
stringEnumName           = enumName
integerEnumDeclaration   = "{" [ integerEnumElement
                           *( "," integerEnumElement) ] "}"
stringEnumDeclaration    = "{" [ stringEnumElement
                           *( "," stringEnumElement) ] "}"
integerEnumElement       = [ qualifierList ] enumLiteral "=" integerValue
stringEnumElement        = [ qualifierList ] enumLiteral [ "=" stringValue ]
enumLiteral              = IDENTIFIER
ENUMERATION              = "enumeration"         ; keyword: case insensitive
```

The `integerEnumElement` rule states that integer enumeration elements must have explicit and unique integer values as defined in DSP0004. There are two reasons for the requirement to explicitly assign values to integer enumeration values:

- The enumeration values can be declared in any order and, unlike in string enumerations, their value cannot be defaulted
- The derived enumerations can define enumeration values, which fill gaps left in their super-enumeration(s)

The `stringEnumElement` rule states that the values of string enumeration elements are optional. If not declared the value of a string enumeration value is assigned the name of the value itself.

The `integerEnumElement` and the `stringEnumElement` rules also state that enumeration values can be qualified. This is most commonly used to add the Description qualifier to individual iteration elements, but the Experimental and Deprecated qualifiers can be also used (see DSP0004 clause 7).

As defined in DSP0004, enumerations can be defined at the schema level or inside declarations of structures, classes, or associations. Enumerations defined inside those other types are referred to as the "local" enumeration declarations. All other enumerations are defined at the schema level. The names of schema level enumerations shall conform to the `schemaQualifiedName` format rule, which requires that their names begin with the name of the scheme followed by the underscore (U+005F).

The GOLF schema contains a number of enumeration declarations. An example of local string enumeration is MonthsEnum, which is defined in the structure GOLF_Date.

It is a string enumeration, and string enumerations do not require that values are assigned. If a value is not assigned, it is assumed to be identical to the name, so in the example above the value of January is "January".

The GOLF_StatesEnum is an example of a schema level string enumeration that assigns explicit values, which are different than the enumeration names.

The following are two schema level integer enumerations GOLF_ProfessionalStatusEnum and GOLF_MemberStatusEnum) that derive from each other.

```
// ==================================================================
//  GOLF_ProfessionalStatusEnum
// ==================================================================
enumeration GOLF_ProfessionalStatusEnum : uint16
{
   Professional = 6,
   SponsoredProfessional = 7
};

// ==================================================================
//  GOLF_MemberStatusEnum
// ==================================================================
enumeration GOLF_MemberStatusEnum : GOLF_ProfessionalStatusEnum
{
    Basic = 0,
    Extended = 1,
    VP = 2,
};
```

The example may look a bit contrived, but it illustrates two important points:

- The values of the integer enumeration values can be defined in any order. In the example the base enumeration GOLF_ProfessionalStatusEnum defines values 6 and 7, while the derived enumeration GOLF_MemberStatusEnum adds values 0, 1, and 2.
- When the type of an enumeration property is overridden in a subclass, the new type can only be the supertype of the overridden type. This is illustrated by the definitions of the GOLF_ClubMember and GOLF_Professional classes and described in the subclause 5.6.3.3 of DSP0004. The reason for this restriction is that an overriding property in a subclass must constrain its values to the same set or a subset of the values of the overridden property.

In addition to the grammar rules stated above a MOF compiler shall verify the integrity of enumeration declarations using the applicable CIM Metamodel constraints, which are stated as OCL constraints in subclause 5.6.1 of DSP0004 and listed in ANNEX G of that document.

7.3.2 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. A structure can be also used as a base for a class, in which case the class derived from the structure inherits all of its features.

The `structureDeclaration` MOF grammar rule corresponds to the Structure CIM metaelement defined in DSP0004 and shall conform to the following set of ABNF rules (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

```
structureDeclaration    = [ qualifierList ] STRUCTURE structureName
                          [ superStructure ]
                          "{" *structureFeature "}" ";"
structureName           = elementName
superStructure          = ":" structureName
structureFeature        = structureDeclaration /   ; local structure
                          enumDeclaration /        ; local enumeration
                          propertyDeclaration
STRUCTURE               = "structure"              ; keyword: case insensitive
```

Structure is a, possibly empty, collection of properties, local structure declarations, and local enumeration declarations. A structure can derive from another structure (see the *superType* reflective association of the Type CIM metaelement in DSP0004). A structure can be declared at the schema level, and therefore be globally visible to all other structures, classes and associations, or its declaration can be local to a structure, a class or an association declaration and be visible only in that structure, class, or association and its derived types.

The `propertyDeclaration` in MOF corresponds to the Property CIM metaelement defined in DSP0004 and shall conform to the following ABNF rules (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

```
propertyDeclaration     = [ qualifierList ] ( primitivePropertyDeclaration /
                          complexPropertyDeclaration /
                          enumPropertyDeclaration /
                          referencePropertyDeclaration ) ";"
primitivePropertyDeclaration = primitiveType propertyName [ array ]
                          [ "=" primitiveTypeValue ]
complexPropertyDeclaration = structureOrClassName propertyName [ array ]
                          [ "=" ( complexTypeValue / aliasIdentifier ) ]
enumPropertyDeclaration = enumName propertyName [ array ]
                          [ "=" enumTypeValue ]
referencePropertyDeclaration = classReference propertyName [ array ]
                          [ "="  referenceTypeValue ]
array                   = "[" "]"
propertyName            = IDENTIFIER
structureOrClassName    = structureName / className
```

The GOLF_Date is an example of a schema-level structure with locally defined enumeration and three properties. All three properties have default values that set the default value of the entire structure to January 1, 2000.

The general form of a reference to an enumeration value is qualified with the name of the enumeration, as it is shown in the example of the default value of the Month property of the GOLF_Date structure.

```
GOLF_MonthsEnum Month = MonthsEnum.January
```

However when the enumeration type is implied, as in the example above, a reference to enumeration value can be simplified by omitting the enumeration name.

```
GOLF_MonthsEnum Month = January
```

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

An example of a local structure is Sponsor, which is defined in the GOLF_Professional class. It can be used only in the GOLF_Professional class or a class that derives from it.

In addition to the grammar rules stated above, a MOF compiler shall verify the integrity of structure declarations by using the applicable CIM Metamodel constraints, which are stated as OCL constraints in clause 6 of DSP0004 and listed in ANNEX G of that document.

7.3.3 Class declaration

A class defines 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 also define methods that do not belong to instances of the class, but to the class itself.

In the CIM Metamodel the Class metaelement derives from the Structure metaelement, so like a structure a class can define local structures and enumerations that can be used in that class or its subclasses.

The `classDeclaration` MOF grammar rule corresponds to the Class CIM metaelement defined in DSP0004, and shall conform to the following ABNF rules (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

```
classDeclaration        = [ qualifierList ] CLASS className [ superClass ]
                          "{" *classFeature "}" ";"
className               = elementName
superClass              = ":" className
classFeature            = structureFeature /
                          methodDeclaration
CLASS                   = "class"               ; keyword: case insensitive
```

The `propertyDeclaration` rule is also described in 7.3.2.

The `methodDeclaration` rule corresponds to the Method CIM metaelement defined in DSP0004, and shall conform to the following ABNF rules (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

```
methodDeclaration       = [ qualifierList ]
                          ( ( returnDataType [ array ] ) / VOID )  methodName
                          "(" [ parameterList ] ")" ";"
returnDataType          = primitiveType /
                          structureOrClassName /
                          enumName /
                          classReference
methodName              = IDENTIFIER
classReference          = DT_REFERENCE
VOID                    = "void"                ; keyword: case insensitive
parameterList           = parameterDeclaration *( "," parameterDeclaration )
```

A method can have zero or more parameters. The `parameterDeclaration` MOF grammar rule corresponds to the Parameter CIM metaelement in DSP0004, and it shall conform to the following ABNF rules (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

```
parameterDeclaration    = [ qualifierList ] ( primitiveParamDeclaration /
                          complexParamDeclaration /
                          enumParamDeclaration /
                          referenceParamDeclaration )
primitiveParamDeclaration = primitiveType parameterName [ array ]
                          [ "="  primitiveTypeValue ]
complexParamDeclaration = structureOrClassName parameterName [ array ]
                          [ "=" ( complexTypeValue / aliasIdentifier ) ]
enumParamDeclaration    = enumName parameterName [ array ]
                          [ "=" enumTypeValue ]
referenceParamDeclaration = classReference parameterName [ array ]
                          [ "="  referenceTypeValue ]
parameterName           = IDENTIFIER
```

A class may define two kinds of methods:

- Instance methods, which are invoked on an instance and receive that instance as an additional/implied argument (a concept similar to the "this" method argument in dynamic programming languages
- Static methods, designated with the Static qualifier, which can be invoked on an instance of the class or the class, but when invoked on the instance do not get that instance as an additional argument

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 inherits the properties, enumerations, structures of that super-structure.

A class may be designated as abstract by specifying the Abstract qualifier. An abstract class cannot be separately instantiated, but can be the superclass of non-abstract classes that can have instances (see the Class CIM metaelement and the Abstract qualifier in DSP0004 for more details). The GOLF_Base class is an example of an abstract class.

Non-abstract classes can have one or more key properties. A key property is specified with the Key qualifier (see the Property CIM metaelement and the Key qualifier in DSP0004 for more details). The key properties of a class instance collectively provide a unique identifier for the class instance within a namespace.

The InstanceID property of the GOLF_Base class is an example of a key property. A key property should be of type string, although other primitive types can be used, and must have the Key qualifier. The key property is used by class implementations to uniquely identify instances.

The parameter Status in the method GetNumberOfProfessionals of the GOLF_Professional class illustrates parameter default values. CIM v3 introduces the ability to define default values for method parameters (see the `primitiveParamDeclaration`, `structureParamDeclaration`, `enumParamDeclaration`, `classParamDeclaration` and `referenceParamDeclaration` MOF grammar rules).

The second parameter of the GetNumberOfProfessionals method has the default value MemberStatusEnum.Professional. The parameter default values have been introduced to support method extensions. The idea of the method extensions is as follows:

- A derived class may override a method and add a new parameter.
- The added parameter is declared with a default value.
- A client written against 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 the example simple. However the GetNumberOfProfessionals method can be called with all three arguments, or only with the NoOfPros and Club arguments.

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

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

In addition to the grammar rules stated above, a MOF compiler shall verify the integrity of class declarations using the applicable CIM Metamodel constraints, which are stated as OCL constraints in clause 5.6.7 of DSP0004 and listed in ANNEX G of that document.

7.3.4 Association declaration

An association represents a relationship between two or more classes. The associated classes are specified by the reference properties of the association. Within an association instance each reference property refers to one instance of the referenced class or its subclass. An association instance is the relationship between all referenced class instances.

The `associationDeclaration` MOF grammar rule corresponds to the Association CIM metaelement defined in DSP0004, and shall conform to the following ABNF rules (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

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

In the CIM Metamodel the Association metaelement derives from Class metaelement, and is structurally identical to Class. However an association declaration

- must have at least two scalar reference properties, and
- each reference property represents a role in the association.

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

The multiplicity of the association ends can be defined using the Max and Min qualifiers (see the discussion of associations in subclause 6.2.2 of DSP0004).

In addition to the grammar rules stated above a MOF compiler shall verify the integrity of association declarations using the applicable CIM Metamodel constraints, which are stated as OCL constraints in clause 6 of DSP0004 and listed in ANNEX G of that document.

7.3.5 Primitive types declarations

CIM defines the following set of primitive data types:

- numeric
  - integer
    - signedInteger
      - sint8, sint16,sint32, sint64, s
    - unsignedIntegers
      - uint8, uint16, uint32, uint64
  - real
    - real32, real64
- string
- datetime
- boolean, and
- octetstring

Each MOF primitive data type corresponds to a CIM Metamodel element derived from the PrimitiveType metaelement as defined in DSP0004. A MOF primitive data type shall conform to the following `primitiveType` ABNF rule (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

```
primitiveType           = DT_Integer /
                          DT_Real /
                          DT_STRING /
                          DT_DATETIME /
                          DT_BOOLEAN /
                          DT_OCTETSTRING
DT_Integer              = DT_UnsignedInteger /
                          DT_SignedInteger
DT_Real                 = DT_REAL32 /
                          DT_REAL64 /
DT_UnsignedInteger      = DT_UINT8 /
                          DT_UINT16 /
                          DT_UINT32 /
                          DT_UINT64
DT_SignedInteger        = DT_SINT8 /
                          DT_SINT16 /
                          DT_SINT32 /
                          DT_SINT64
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_SINT8                = "sint8"               ; keyword: case insensitive
DT_SINT16               = "sint16"              ; keyword: case insensitive
DT_SINT32               = "sint32"              ; keyword: case insensitive
DT_SINT64               = "sint64"              ; keyword: case insensitive
DT_REAL32               = "real32"              ; keyword: case insensitive
DT_REAL64               = "real64"              ; keyword: case insensitive
DT_STRING               = "string"              ; keyword: case insensitive
DT_DATETIME             = "datetime"            ; keyword: case insensitive
DT_BOOLEAN              = "boolean"             ; keyword: case insensitive
DT_OCTETSTRING          = "octetstring"         ; keyword: case insensitive
```

The primitive types are used in the declarations of

- Qualifiers types
- Properties
- Enumerations

- Method parameters
- Method results

7.3.6 Reference type declaration

The reference type corresponds to the ReferenceType CIM metaelement. A declaration of a reference type shall conform to ABNF rule `DT_REFERENCE` (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

```
DT_REFERENCE            = className REF
REF                     = "ref"                 ; keyword: case insensitive
```

ANNEX A
(normative)

MOF grammar description

The grammar is defined by using the ABNF notation described in RFC5234.

The definition uses the following conventions:

- Punctuation terminals like `";"` are shown verbatim.
- Terminal symbols are spelled in CAPITAL letters 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 CIM Metamodel constraints are expected to be checked over an in-memory MOF representation (the ASTs) after all MOF files in a compilation unit have been parsed. For example, the constraint that a property in a derived class must not have the same name as an inherited property unless it overrides that property (has the Override qualifier) is not encoded in the grammar. Similarly the default values of qualifier definitions are lexically permissive to keep parsing simple.

The MOF compiler developers should assume that unless explicitly stated otherwise, the terminal symbols are separated by whitespace (see 5.2).

The MOF v3 grammar is written with the objective to minimize the differences between this version the MOF v2 version. The three differences that the MOF compiler developer will have to take into account are:

- The qualifier declaration has a different grammar
- Arbitrary UCS characters are no longer supported as identifiers
- Octetstring values do not have the length bytes at the beginning
- Fixed size arrays are no longer supported
- The char16 datatype has been removed

A.1 Value definitions

In MOF a value, or an array of values, can be specified as:

- default value of a property or a method parameter
- default value of a qualifier type declaration
- qualifier value
- value of a property in a specification of a structure value or class or association instance

MOF divides values into four categories:

- Primitive type values
- Complex type values
- Enumeration type values
- Reference type values

The `primitiveTypeValue` MOF grammar rule corresponds to the LiteralSpecification CIM metaelement and represents a single value, or an array of values of the predefined primitive types (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section).

```
primitiveTypeValue      = literalValue / literalValueArray
literalValueArray       = "{" [ literalValue *( "," literalValue ) ] "}"
literalValue            = integerValue / realValue /
                          stringValue / octetStringValue
                          booleanValue /
                          nullValue /
                          dateTimeValue
```

The MOF grammar rules for the different types of literals are defined in A.16.

The `complexTypeValue` MOF grammar rule corresponds to the ComplexValue CIM metaelement, and shall conform to the following ABNF rules (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

```
complexTypeValue        = complexValue / complexValueArray
complexValueArray       = "{" [ complexValue *( "," complexValue) ] "}"
complexValue            = ( INSTANCE / VALUE ) OF
                          ( structureName / className / associationName )
                          [ alias ]  propertyValueList
propertyValueList       = "{" *propertySlot "}"
propertySlot            = propertyName "=" propertyValue ";"
propertyValue           = primitiveTypeValue / complexTypeValue /
                          referenceTypeValue / enumTypeValue
alias                   = AS aliasIdentifier
INSTANCE                = "instance"           ; keyword: case insensitive
VALUE                   = "value"              ; keyword: case insensitive
AS                      = "as"                 ; keyword: case insensitive
OF                      = "of"                 ; keyword: case insensitive
```

A complex value specification can start with one of two keywords; `"instance"` or `"value"`.

The keyword `"value"` corresponds to the StructureValue CIM metaelement. It shall be used to define a value of a structure, class, or association that only will be used as the

- value of complex property in instances of a class or association, or in structure value
- default value of a property
- default value of a method parameter

The keyword "instance" corresponds to the InstanceSpecification CIM metaelement and shall be used to define an instance of a class or association.

The JohnDoe_mof is an example of an instance value that represents a person with the first name "John" and the last name "Doe".

Values of structures can be defined in two ways:

- By inlining them inside the owner class or structure instance. An example is the value of LastPaymentDate property, or
- By defining them separately and giving them aliases. Examples are $JohnDoesPhoneNo and $JohnDoesStartDate, which are first predefined and then used in the definition of the John Doe instance.

The rules for the representation of the values of schema elements of type enumeration or reference are described in A.18 and A.19 respectively.

In addition to the grammar rules stated above a MOF compiler shall verify the integrity of value description statements by using the applicable CIM Metamodel constraints, which are stated as OCL constraints in clause 6 of DSP0004 and listed in ANNEX G of that document.

A.2 MOF specification

A MOF specification defines one or more schema elements and is derived by a MOF compiler from a MOF compilation unit. A MOF specification shall conform to ABNF rule `mofSpecification` (whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section):

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

A.3 Compiler directive

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
compilerDirective       = PRAGMA ( pragmaName / standardPragmaName )
                          "(" pragmaParameter ")"
pragmaName              = IDENTIFIER
standardPragmaName      = INCLUDE
pragmaParameter         = stringValue           ; if the pragma is INCLUDE,
                                                ; the parameter value
                                                ; shall represent a relative
                                                ; or full file path
PRAGMA                  = "#pragma"             ; keyword: case insensitive
INCLUDE                 = "include"             ; keyword: case insensitive
```

A.4 Structure declaration

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

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
structureDeclaration    = [ qualifierList ] STRUCTURE structureName
                          [ superstructure ]
                          "{" *structureFeature "}" ";"
structureName           = elementName
superStructure          = ":" structureName
structureFeature        = structureDeclaration /  ; local structure
                          enumDeclaration /     ; local enumeration
                          propertyDeclaration
STRUCTURE               = "structure"           ; keyword: case insensitive
```

A.5 Class declaration

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
classDeclaration        = [ qualifierList ] CLASS className [ superClass ]
                          "{" *classFeature "}" ";"
className               = elementName
superClass              = ":" className
classFeature            = structureFeature /
                          methodDeclaration
CLASS                   = "class"               ; keyword: case insensitive
```

A.6 Association declaration

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

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

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

A.7 Enumeration declaration

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.

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
enumDeclaration         = enumTypeHeader
                          enumName ":" enumTypeDeclaration ";"
enumTypeHeader          = [ qualifierList ] ENUMERATION
enumName                = elementName
enumTypeDeclaration     = ( DT_Integer / integerEnumName )
                          integerEnumDeclaration /
                          ( DT_STRING / stringEnumName )
                          stringEnumDeclaration
integerEnumName         = enumName
stringEnumName          = enumName
integerEnumDeclaration  = "{" [ integerEnumElement
                          *( "," integerEnumElement ) ] "}"
stringEnumDeclaration   = "{" [ stringEnumElement
                          *( "," stringEnumElement ) ] "}"
integerEnumElement      = [ qualifierList ] enumLiteral "=" integerValue
stringEnumElement       = [ qualifierList ] enumLiteral [ "=" stringValue ]
enumLiteral             = IDENTIFIER
ENUMERATION             = "enumeration"         ; keyword: case insensitive
```

A.8 Qualifier type declaration

Notice that qualifiers can be qualified themselves. This is mainly to allow for describing and deprecating qualifiers.

Because DSP0004 in CIM v3 the qualifier flavor has been replaced with qualifier policy, the MOF v2 qualifier declarations have to be converted to MOF v3 before parsing.

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
qualifierTypeDeclaration = [ qualifierList ]
                          QUALIFIER  qualifierName ":" qualifierType
                          qualifierScope [ qualifierPolicy ] ";"
qualifierName           = elementName
qualifierType           = primitiveQualifierType / enumQualiferType
primitiveQualifierType  = primitiveType [ array ]
                          [ "=" primitiveTypeValue ] ";"
enumQualiferType        = enumName [ array ] "=" enumTypeValue ";"
qualifierScope          = SCOPE "(" ANY / scopeKindList ")"
```

```
qualifierPolicy         = POLICY "(" policyKind ")"
policyKind              = DISABLEOVERRIDE /
                          ENABLEOVERRIDE /
                          RESTRICTED
scopeKindList           = scopeKind  *( "," scopeKind )
scopeKind               = STRUCTURE / CLASS / ASSOCIATION /
                          ENUMERATION / ENUMERATIONVALUE /
                          PROPERTY / REFPROPERTY /
                          METHOD / PARAMETER /
                          QUALIFIERTYPE
SCOPE                   = "scope"               ; keyword: case insensitive
ANY                     = "any"                 ; keyword: case insensitive
POLICY                  = "policy"              ; keyword: case insensitive
ENABLEOVERRIDE          = "enableoverride"      ; keyword: case insensitive
DISABLEOVERRIDE         = "disableoverride"     ; keyword: case insensitive
RESTRICTED              = "restricted"          ; keyword: case insensitive
ENUMERATIONVALUE        = "enumerationvalue"    ; keyword: case insensitive
PROPERTY                = "property"            ; keyword: case insensitive
REFPROPETY              = "reference"           ; keyword: case insensitive
METHOD                  = "method"              ; keyword: case insensitive
PARAMETER               = "parameter"           ; keyword: case insensitive
QUALIFIERTYPE           = "qualifiertype"       ; keyword: case insensitive
```

A.9 Qualifier list

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
qualifierList           = "[" qualifierValue *( "," qualifierValue ) "]"
qualifierValue          = qualifierName [ qualifierValueInitializer /
                          qualiferValueArrayInitializer  ]
qualifierValueInitializer = "(" literalValue ")"
qualiferValueArrayInitializer = "{" literalValue *( "," literalValue ) "}"
```

A.10 Property declaration

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
propertyDeclaration     = [ qualifierList ] ( primitivePropertyDeclaration /
                          complexPropertyDeclaration /
                          enumPropertyDeclaration /
                          referencePropertyDeclaration ) ";"
primitivePropertyDeclaration = primitiveType propertyName [ array ]
                          [ "=" primitiveTypeValue ]
complexPropertyDeclaration = structureOrClassName propertyName [ array ]
                          [ "=" ( complexTypeValue / aliasIdentifier ) ]
enumPropertyDeclaration = enumName propertyName [ array ]
                          [  "=" enumTypeValue ]
referencePropertyDeclaration = classReference propertyName [ array ]
                          [ "="  referenceTypeValue ]
array                   = "[" "]"
propertyName            = IDENTIFIER
structureOrClassName    = IDENTIFIER
REF                     = "ref"                 ; keyword: case insensitive
```

A.11 Method declaration

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
methodDeclaration       = [ qualifierList ] ( ( returnDataType [ array ] ) /
                          VOID )  methodName
                          "(" [ parameterList ] ")" ";"
returnDataType          = primitiveType /
                          structureOrClassName /
                          enumName /
                          classReference
methodName              = IDENTIFIER
classReference          = DT_REFERENCE
VOID                    = "void"                ; keyword: case insensitive
parameterList           = parameterDeclaration *( "," parameterDeclaration )
```

A.12 Parameter declaration

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
parameterDeclaration    = [ qualifierList ] ( primitiveParamDeclaration /
                          complexParamDeclaration /
                          enumParamDeclaration /
                          referenceParamDeclaration )
primitiveParamDeclaration = primitiveType parameterName [ array ]
                          [ "="  primitiveTypeValue ]
complexParamDeclaration = structureOrClassName parameterName [ array ]
                          [ "=" ( complexTypeValue / aliasIdentifier ) ]
enumParamDeclaration    = enumName parameterName [ array ]
                          [ "=" enumValue ]
referenceParamDeclaration = classReference parameterName [ array ]
                          [ "="  referenceTypeValue ]
parameterName           = IDENTIFIER
```

A.13 Names

MOF names are identifiers with the format defined by the `IDENTIFIER` rule.

No whitespace is allowed between the elements of the rules in this ABNF section.

```
IDENTIFIER              = firstIdentifierChar *( nextIdentifierChar )
firstIdentifierChar     = UPPERALPHA / LOWERALPHA / UNDERSCORE
nextIdentifierChar      = firstIdentifierChar / decimalDigit
elementName             = localName / schemaQualifiedName
localName               = IDENTIFIER
```

A.13.1 Schema-qualified name

To assure schema level uniqueness of the names of structures, classes, associations, enumerations, and qualifiers, CIM follows a naming convention referred to as the schema-qualified names. A schema-qualified name starts with a globally unique, preferably registered, string associated with a company, business, or organization followed by the underscore "_". That unique string is referred to as the schema name. The schemaQualifiedName MOF rule defines the format of the schema-qualified names.

No whitespace is allowed between the elements of the rules in this ABNF section.

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

A.13.2 Alias identifier

No whitespace is allowed between the elements of this rule.

```
aliasIdentifier         = "$" IDENTIFIER
```

A.13.3 Namespace name

The format of the names of namespaces is defined by the `namespaceName` MOF rule.

No whitespace is allowed between the elements of this rule.

```
namespaceName           = IDENTIFIER *( "/" IDENTIFIER )
```

A.14 Complex type value

The grammar is not attempting to verify that the type of the property value is consistent with the type of the property to which the value is assigned. For example, if a property type is a structure containing a string and an integer, its value shall be an instance of that structure with a value for its two properties.

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
complexTypeValue        = complexValue / complexValueArray
complexValueArray       = "{" [ complexValue *( "," complexValue)  ] "}"
complexValue            = ( INSTANCE / VALUE ) [OF]
                          ( structureName / className / associationName )
                          [ alias ]  propertyValueList ";"
propertyValueList       = "{" *propertySlot "}"
propertySlot            = propertyName "=" propertyValue ";"
propertyValue           = primitiveTypeValue / complexTypeValue /
                          referenceTypeValue / enumTypeValue
alias                   = AS aliasIdentifier
INSTANCE                = "instance"            ; keyword: case insensitive
VALUE                   = "value"               ; keyword: case insensitive
AS                      = "as"                  ; keyword: case insensitive
OF                      = "of"                  ; keyword: case insensitive
```

A.15 Primitive data types

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
primitiveType           = DT_Integer /
                          DT_Real /
                          DT_STRING /
                          DT_DATETIME /
                          DT_BOOLEAN /
                          DT_OCTETSTRING
DT_Integer              = DT_UnsignedInteger /
                          DT_SignedInteger
DT_Real                 = DT_REAL32 /
                          DT_REAL64 /
DT_UnsignedInteger      = DT_UINT8 /
                          DT_UINT16 /
                          DT_UINT32 /
                          DT_UINT64
DT_SignedInteger        = DT_SINT8 /
                          DT_SINT16 /
                          DT_SINT32 /
                          DT_SINT64
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_SINT8                = "sint8"               ; keyword: case insensitive
DT_SINT16               = "sint16"              ; keyword: case insensitive
DT_SINT32               = "sint32"              ; keyword: case insensitive
DT_SINT64               = "sint64"              ; keyword: case insensitive
DT_REAL32               = "real32"              ; keyword: case insensitive
DT_REAL64               = "real64"              ; keyword: case insensitive
DT_STRING               = "string"              ; keyword: case insensitive
DT_DATETIME             = "datetime"            ; keyword: case insensitive
DT_BOOLEAN              = "boolean"             ; keyword: case insensitive
DT_OCTETSTRING          = "octetstring"         ; keyword: case insensitive
```

A.16 Reference data type

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
DT_REFERENCE            = className REF
REF                     = "ref"                 ; keyword: case insensitive
```

A.17 Primitive type values

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
primitiveTypeValue      = literalValue / literalValueArray
literalValueArray       = "{" [ literalValue *( "," literalValue ) ] "}"
literalValue            = integerValue /
                          realValue /
                          dateTimeValue /
                          stringValue /
                          booleanValue /
                          octetStringValue /
                          nullValue
```

A.17.1 Integer value

No whitespace is allowed between the elements of the rules in this ABNF section.

```
integerValue            = binaryValue / octalValue / hexValue / decimalValue
binaryValue             = [ "+" / "-" ] 1*binaryDigit ( "b" / "B" )
binaryDigit             = "0" / "1"
octalValue              = [ "+" / "-" ] unsignedOctalValue
unsignedOctalValue      = "0" 1*octalDigit
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            = [ "+" / "-" ] unsignedDecimalValue
unsignedDecimalValue    = positiveDecimalDigit *decimalDigit
```

A.17.2 Real value

No whitespace is allowed between the elements of the rules in this ABNF section.

```
realValue               = [ "+" / "-" ] *decimalDigit "." 1*decimalDigit
                          [ ( "e" / "E" ) [ "+" / "-" ] 1*decimalDigit ]
decimalDigit            = "0" / positiveDecimalDigit
positiveDecimalDigit    = 1"..."9"
```

A.17.3 String values

Unless explicitly specified via ABNF rule `WS`, no whitespace is allowed between the elements of the rules in this ABNF section.

```
stringValue             = DOUBLEQUOTE *stringChar DOUBLEQUOTE
                          *( *WS DOUBLEQUOTE *stringChar DOUBLEQUOTE )
stringChar              = stringUCSchar / stringEscapeSequence
```

```
stringUCSchar           = U+0020...U+0021 / U+0023...U+D7FF /
                          U+E000...U+FFFD / U+10000...U+10FFFF
                          ; Note that these UCS characters can be
                          ; represented in XML without any escaping
                          ; (see W3C XML).
stringEscapeSequence    = BACKSLASH ( BACKSLASH / DOUBLEQUOTE / SINGLEQUOTE /
                          BACKSPACE_ESC / TAB_ESC / LINEFEED_ESC /
                          FORMFEED_ESC / CARRIAGERETURN_ESC /
                          escapedUCSchar )
BACKSPACE_ESC           = "b"        ; escape for back space (U+0008)
TAB_ESC                 = "t"        ; escape for horizontal tab (U+0009)
LINEFEED_ESC            = "n"        ; escape for line feed (U+000A)
FORMFEED_ESC            = "f"        ; escape for form feed (U+000C)
CARRIAGERETURN_ESC      = "r"        ; escape for carriage return (U+000D)
escapedUCSchar          = ( "x" / "X" ) 1*6( hexDigit )  ; escaped UCS
                          ; character with a UCS code position that is
                          ; the numeric value of the hex number
```

A.17.4 Special characters

The following special characters are used in other ABNF rules in this specification:

```
BACKSLASH               = U+005C                ; \
DOUBLEQUOTE             = U+0022                ; "
SINGLEQUOTE             = U+0027                ; '
UPPERALPHA              = U+0041...U+005A       ; A ... Z
LOWERALPHA              = U+0061...U+007A       ; a ... z
UNDERSCORE              = U+005F                ; _
```

A.17.5 OctetString value

Unless explicitly specified via ABNF rule `WS`, no whitespace is allowed between the elements of the rules in this ABNF section.

```
octetStringValue        = DOUBLEQUOTE "0x" *( octetStringElementValue )
                          DOUBLEQUOTE
                          *( *WS DOUBLEQUOTE *( octetStringElementValue )
                          DOUBLEQUOTE )
octetStringElementValue = 2(hexDigit)
```

A.17.6 Boolean value

```
booleanValue             = TRUE / FALSE
FALSE                    = "false"               ; keyword: case insensitive
TRUE                     = "true"                ; keyword: case insensitive
```

A.17.7 Null value

```
nullValue                = NULL
NULL                     = "null"                ; keyword: case insensitive
                         ; second
```

A.17.8 File path

The `filePath` ABNF rule defines the format of the file path used as the string value in the `INCLUDE` compiler directive (see Table 1).

The escape mechanisms defined for the stringValue ABNF rule apply. For example, backslash characters in file paths must be escaped.

A file path can be either a relative path or a full path. The relative path is in relationship to the directory of the file in which the `INCLUDE` compiler directive is found. File paths are subject to platform-specific restrictions on the character set used in directory names and on the length of single directory names and the entire file path.

MOF compilers shall support both forward and backward slashes in path delimiters, including a mix of both.

If the platform has restrictions with respect to these path delimiters, the MOF compiler shall transform the path delimiters to what the platform supports.

No whitespace is allowed between the elements of the rules in this ABNF section.

```
filePath                 = [absoluteFilePrefix] relativeFilePath
relativeFilePath         = IDENTIFIER *( pathDelimiter IDENTIFIER)
pathDelimiter            = "/" / "\"      absoluteFilePrefix  = rootDirectory /
driveLetter
rootDirectory            = pathDelimiter
driveLetter              = UPPERALPHA ":" [pathDelimiter]
```

A.18 Enum type value

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
enumTypeValue            = enumValue / enumValueArray
enumValueArray           = "{" [ enumName *( "," enumName ) ] "}"
enumValue                = [ enumName "." ] enumLiteral
enumLiteral              = IDENTIFIER
```

A.19 Reference type value

ReferenceTypeValues enable a protocol agnostic serialization of a reference.

Whitespace as defined in 5.2 is allowed between the elements of the rules in this ABNF section.

```
referenceTypeValue      = referenceValue / referenceValueArray
referenceValueArray     = "{" [ objectPathValue *( "," objectPathValue ) ]
                          "}"
```

No whitespace is allowed between the elements of the rules in this ABNF section.

```
objectPathValue         = [namespacePath ":"] instanceId
namespacePath           = [serverPath] namespaceName
; Note: The production rules for host and port are defined in IETF
; RFC 3986 (Uniform Resource Identifiers (URI): Generic Syntax).
serverPath              = (host / LOCALHOST) [ ":" port] "/"
LOCALHOST               = "localhost"             ; Case insensitive
instanceId              = className "." instanceKeyValue
instanceKeyValue        = keyValue  *( "," keyValue )
keyValue                = propertyName "=" literalValue
```

ANNEX B
(normative)

MOF keywords

Below are the MOF keywords, listed in alphabetical order.

```
#pragma

any
as
association

boolean

class

datetime
disableoverride

enableoverride
enumeration
enumerationvalue

false
flavor

include
instance

method

null

octetstring
of

parameter
property

qualifier

real32
real64
ref
restricted

scope
sint8
sint16
sint32
sint64
string
structure

true

uint8
uint16
uint32
uint64

value
void
```

ANNEX C
(informative)

Datetime values

The representation of time-related values is defined in DSP0004, clause 5.5.1. The values of the datetime primitive type have one of two formats:

- `timestampValue`, which represents a specific moment in time
- `durationValue`, which represents the length of a time period

No whitespace is allowed between the elements of the rules in this ABNF section.

```
datetimeValue            = timestampValue / durationValue
timestampValue           = DOUBLEQUOTE yearMonthDayHourMinSec "." microseconds
                           ( "+" / "-" ) datetimeTimezone DOUBLEQUOTE
yearMonthDayHourMinSec   = 4Y 2M 2D 2h 2m 2s /
                           4Y 2M 2D 2h 2m 2"*" /
                           4Y 2M 2D 2h 4"*" /
                           4Y 2M 2D 6"*" /
                           4Y 2M 8"*" /
                           4Y 10"*" /
                           14"*"
datetimeTimezone         = 3m
durationValue            = DOUBLEQUOTE dayHourMinSec "." microseconds
                           ":000" DOUBLEQUOTE
dayHourMinSec            = 8D 2h 2m 2s /
                           8D 2h 2m 2"*" /
                           8D 2h 4"*" /
                           8D 6"*" /
                           14"*"
microseconds             = 6decimalDigit /
                           5decimalDigit "*" /
                           4decimalDigit 2"*" /
                           3decimalDigit 3"*" /
                           2decimalDigit 4"*" /
                           decimalDigit 5"*" /
                           6"*"
Y                        = decimalDigit          ; year
M                        = decimalDigit          ; month
```

```
D                       = decimalDigit          ; day
h                       = decimalDigit          ; hour
m                       = decimalDigit          ; minute
s                       = decimalDigit          ; second
```

ANNEX D
(informative)

Programmatic units

The following rules define the string representation of a unit of measurement for programmatic access. Programmatic unit is described in detail and exemplified in ANNEX D of DSP0004.

The following special characters are used only in programmatic units.

```
HYPHEN                  = U+002D                  ; -
CARET                   = U+005E                  ; ^
COLON                   = U+003A                  ; :
PARENS                  = U+0028 / U+0029         ; ( and )
SPACE                   = U+0020                  ; " "
```

A programmatic unit can be used as a

- value of the PUnit qualifier
- value of a string typed model element qualified with the boolean IsPUnit qualifier

Unless specified via the ABNF rule `SPACE`, no whitespace is allowed between the elements of the rules in this ABNF section.

```
programmaticUnitValue   = DOUBLEQUOTE programmaticUnit DOUBLEQUOTE
programmaticUnit        = [HYPHEN] *SPACE unitElement
                          *( *SPACE unitOperator *SPACE unitElement )
unitElement             = ( floatingPointNumber / exponentialNumber ) /
                          [ unitPrefix ] baseUnit [ CARET exponent ]
floatingPointNumber     = 1*( decimalDigit)  [ "." ]  *( decimalDigit )
exponentialNumber       = unsignedDecimalValue CARET  exponent
                          ; shall be interpreted as a floating point number
                          ; with the specified decimal base and decimal
                          ; exponent and a mantissa of 1
exponent                = [ HYPHEN ]  unsignedDecimalValue
unsignedDecimalValue    = positiveDecimalDigit *( decimalDigit)
unitOperator            = "*" / "/"
unitPrefix              = decimalPrefix  /  binaryPrefix
                          ; The numeric equivalents of these prefixes shall
                          ; be interpreted as multiplication factors for the
                          ; directly succeeding base unit. In other words,
                          ; if a prefixed base unit is in the denominator
                          ; of the overall programmatic unit, the numeric
                          ; equivalent of that prefix is also in the
                          ; denominator.

; SI decimal prefixes as defined in ISO 1000:1992:
decimalPrefix           = "deca" /              ; 10^1
                          "hecto" /             ; 10^2
                          "kilo" /              ; 10^3
                          "mega" /              ; 10^6
                          "giga" /              ; 10^9
                          "tera" /              ; 10^12
                          "peta" /              ; 10^15
                          "exa" /               ; 10^18
                          "zetta" /             ; 10^21
                          "yotta" /             ; 10^24
                          "deci" /              ; 10^-1
                          "centi" /             ; 10^-2
                          "milli" /             ; 10^-3
                          "micro" /             ; 10^-6
                          "nano" /              ; 10^-9
                          "pico" /              ; 10^-12
                          "femto" /             ; 10^-15
                          "atto" /              ; 10^-18
                          "zepto" /             ; 10^-21
                          "yocto"               ; 10^-24

; IEC binary prefixes as defined in ISO/IEC 80000-13:
binaryPrefix            = "kibi" /              ; 2^10
                          "mebi" /              ; 2^20
                          "gibi" /              ; 2^30
                          "tebi" /              ; 2^40
                          "pebi" /              ; 2^50
                          "exbi" /              ; 2^60
                          "zebi" /              ; 2^70
                          "yobi"                ; 2^80
baseUnit                = unitIdentifier / extensionUnit
                          ; If unitIdentifier begins with a prefix
                          ; (see prefix ABNF rule), the meaning of
                          ; that prefix shall not be changed by the extension
                          ; base unit (examples of this for standard base
                          ; units are "decibel" or "kilogram")
extensionUnit           = orgId COLON unitIdentifier
orgId                   = IDENTIFIER
                          ; org-id shall include a copyrighted, trademarked,
                          ; or otherwise unique name that is owned by the
                          ; business entity that is defining the extension
                          ; unit, or that  is a registered ID assigned to
                          ; the business entity by a recognized global
                          ; authority. org-id shall not begin with a prefix
                          ; (see prefix ABNF rule).
unitIdentifier          = firstUnitChar [ *(unitChar ) lastUnitChar ]
firstUnitChar           = UPPERALPHA / LOWERALPHA / UNDERSCORE
lastUnitChar            = firstUnitChar / decimalDigit / PARENS
unitChar                = lastUnitChar / HYPHEN / SPACE
```

ANNEX E
(informative)

Example MOF specification

The GOLF model has been created only to illustrate the use of MOF, so some of the design choices may not be very appealing. The model contains classes and association shown in the diagram below.

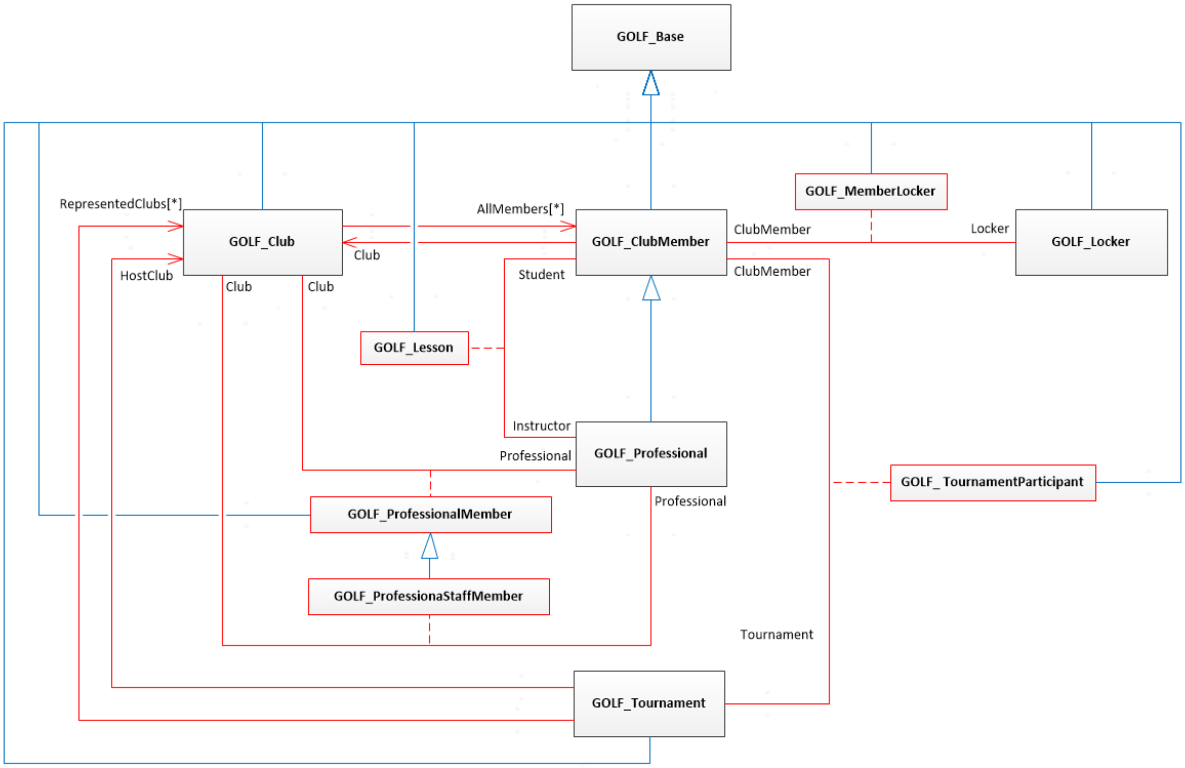

Figure E-1 − Classes and association of the GOLF model

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

E.1 GOLF_Schema.mof

```
// ==================================================================
// Copyright 2012 Distributed Management Task Force, Inc. (DMTF).
// Example 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")
#pragma include ("GOLF_Tournament.mof")
#pragma include ("GOLF_TournamentParticipant.mof")
//
// Schema level 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_MemberStatusEnum.mof")
#pragma include ("GlobalEnums/GOLF_ProfessionalStatusEnum.mof")
#pragma include ("GlobalEnums/GOLF_GOLF_StatesEnum.mof")
//
//  Instances
//
#pragma include ("Instances/JohnDoe.mof")
```

E.2 GOLF_Base.mof

```
// ==================================================================
//  GOLF_Base
// ==================================================================
        [Abstract,
         OCL { "-- the key property cannot be NULL\n"
               "inv: not InstanceId.oclIsUndefined()",
               "-- in the GOLF model the InstanceId must have exactly "
               "10 characters\n"
               "inv: InstanceId.size() = 10" } ]
class GOLF_Base {
// =========================== properties ===========================
        [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 short textual description (one- line string) of the
instance." ),
         MaxLen(64)]
    string Caption = Null;
};
```

E.3 GOLF_Club.mof

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

   GOLF_Address ClubAddress;
   GOLF_PhoneNumber ClubPhoneNo;
   GOLF_PhoneNumber ClubFaxNo;
   string ClubWebSiteURL;

   GOLF_ClubMember REF AllMembers[];

// ============================ methods =============================
   GOLF_ResultCodeEnum AddNonProfessionalMember (
      [In] GOLF_ClubMember newMember
   );
       GOLF_ResultCodeEnum AddProfessionalMember (
          [In] GOLF_Professional newProfessional
   );
   UInt32 GetMembersWithOutstandingFees (
      [In] GOLF_Date referenceDate,
      [Out] GOLF_ClubMember REF lateMembers[]
   );
   GOLF_ResultCodeEnum TerminateMembership (
      [In] GOLF_ClubMember REF memberURI
   );
};
```

E.4 GOLF_ClubMember.mof

```
// ==================================================================
//  GOLF_ClubMember
// ==================================================================
    [Description (
    "Instances of this class represent members of a golf club." ),
    OCL{"-- a member with Basic status may only have one locker\n"
        "inv: Status = MemberStatusEnum.Basic implies not "
       "(GOLF_MemberLocker.Locker->size() > 1)",
        "inv: not MemberPhoneNo.oclIsUndefined()",
        "inv: not Club.oclIsUndefined()" } ]
class GOLF_ClubMember: GOLF_Base {

// =========================== properties ===========================
   string FirstName;
   string LastName;
   GOLF_Club REF Club;
   GOLF_MemberStatusEnum Status;
   GOLF_Date MembershipEstablishedDate;

   real32 MembershipSignUpFee;
   real32 MonthlyFee;
   GOLF_Date LastPaymentDate;

   GOLF_Address MemberAddress;
   GOLF_PhoneNumber MemberPhoneNo;
   string MemberEmailAddress;

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

E.5 GOLF_Professional.mof

```
// ==================================================================
//  GOLF_Professional
// ==================================================================
    [Description("instances of this class represent professional members "
        "of the golf club"),
     OCL{"-- to have the sponsored professional status a member must "
         "have at least one sponsor\n"
         "inv: self.Status = SponsoredProfessional implies "
         "\t self.Sponsors->size() > 0" } ]
class GOLF_Professional : GOLF_ClubMember {
// ======================== local structures ========================
   structure Sponsor {
      string Name,
      GOLF_Date ContractSignedDate;
      real32 ContractAmount;
   };

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

// ============================ methods =============================
       [Static]
   GOLF_ResultCodeEnum GetNumberOfProfessionals (
      [Out] Uint32 NoOfPros,
      [In] GOLF_Club Club,
      [In] ProfessionalStatusEnum Status = Professional
   )
};
```

E.6 GOLF_Locker.mof

```
// ==================================================================
//  GOLF_Locker
// ==================================================================
class GOLF_Locker : GOLF_Base {
   string Location;
   uint16 LockerNo;
   real32 MonthlyRentFee;
};
```

E.7 GOLF_Tournament.mof

```
// ==================================================================
//  GOLF_Tournament
// ==================================================================
    [Description ("Instances of this class represent golf tournaments.")
     OCL {"-- each participant must belong to a represented club\n"
          "inv: self.GOLF_TournamentParticipant.Participant->forAll(p | "
           "self.RepresentedClubs -> includes(p.Club))",
          "-- tournament must be hosted by a club \n"
          "inv: not self.HostClub.oclIsUndefined()" } ]
class GOLF_Tournament: GOLF_Base {
// ======================== local structures ========================
        [OCL {"-- none of the result properties can be undefined or empty \n"
              "inv: not oclIsUndefined(self.ParticipantName) and \n"
              "\t not oclIsUndefined(self.ParticipantGolfClubName) and \n"
              "\t self.FinalPosition > 0)" } ]
    structure IndividualResult {
        string ParticipantName;
        string ParticipantGolfClubName;
        unit32 FinalPosition;
    };

// =========================== properties ===========================
   string TournamentName;
   string HostingClubName;
   GOLF_Address HostingClubAddress;
   GOLF_PhoneNumber HostingClubPhoneNo;
   string HostingClubWebPage;

   GOLF_Date StartDate;
   GOLF_Date EndDate;

   string Sponsors[];

   GOLF_Club REF HostClub;
   GOLF_Club REF RepresentedClubs[];

// ============================ methods =============================
   GOLF_ResultCodeEnum GetResults([Out] IndividualResult results[]);
};
```

E.8 GOLF_MemberLocker.mof

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

E.9 GOLF_Lesson.mof

```
// ==================================================================
//  GOLF_Lesson
// ==================================================================
    [Description ( "Instances of the association represent past and "
        "future golf lessons.",
     OCL {"-- lesson can be given only by a professional who is a member "
          "of the club staff \n"
          "inv: Instructor.GOLF_ProfessionalStaffMember.Club->size() = 1" } ]
association GOLF_Lesson : GOLF_Base {
   GOLF_Professional REF Instructor;
   GOLF_ClubMember REF Student;

   datetime Schedule;
       [Description ( "The duration of the lesson" )]
   datetime Length = "**********60**.******:000";
   string Location;
       [Description ( " Cost of the lesson in US$ ")]
   real32 LessonFee;
};
```

E.10 GOLF_ProfessionalMember.mof

```
// ==================================================================
//  GOLF_ProfessionalMember
// ==================================================================
    [Description (
     "Instances of this association represent club membership "
     "of professional golfers that are not members of the club staff." )
    ]
association GOLF_ProfessionalMember : GOLF_Base {
   GOLF_Professional REF Professional;
   GOLF_Club REF Club;
};
```

E.11 GOLF_ProfessionalStaffMember.mof

```
// ==================================================================
//  GOLF_ ProfessionalStaffMember
// ==================================================================
    [Description ( "Instances of this association represent club membership "
      "of professional golfers who are members of the club staff "
      "and earn a salary." ) ]
association GOLF_ProfessionalStaffMember : GOLF_ProfessionalNonStaffMember {
   GOLF_Professional REF Professional;
   GOLF_Club REF Club;
       [Description ( "Monthly salary in $US" ) ]
    real32 Salary;
};
```

E.12 GOLF_TournamentParticipant.mof

```
// ==================================================================
//  GOLF_ TournamentParticipant
// ==================================================================
    [Description ( "Instances of this association represent golf members of"
        "golf clubs participating in tournaments." ),
     OCL { "-- the club of the participant must be represented in the "
            "tournament \n"
           "inv: Tournament.RepresentedClubs->includes(Participant.Club)" } ]
association GOLF_TournamentParticipant : GOLF_Base {
   GOLF_ClubMember REF Participant;
   GOLF_Tournament REF Tournament;
   uint32 FinalPosition = 0;
};
```

E.13 GOLF_Address.mof

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

E.14 GOLF_Date.mof

```
// ==================================================================
//  GOLF_Date
// ==================================================================
structure GOLF_Date {
// ======================= local enumerations =======================
    enumeration MonthsEnum : String {
       January,
       February,
       March,
       April,
       May,
       June,
       July,
       August,
       September,
       October,
       November,
       December
    };

// =========================== properties ===========================
   uint16 Year = 2000;
   MonthsEnum Month = MonthsEnum.January;
       [MinValue(1), MaxValue(31)]
   uint16 Day = 1;
};
```

E.15 GOLF_PhoneNumber.mof

```
// ==================================================================
//  GOLF_PhoneNumber
// ==================================================================
    [OCL { "inv: AreaCode -> size() = 3",
           "inv: Number->size() = 7" } ]
```

```
structure GOLF_PhoneNumber {
   uint8 AreaCode[];
   uint8 Number[];
};
```

E.16 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
};
```

E.17 GOLF_ProfessionalStatusEnum.mof

```
// ==================================================================
//  GOLF_ProfessionalStatusEnum
// ==================================================================
enumeration GOLF_ProfessionalStatusEnum : uint16
{
   Professional = 6,
   SponsoredProfessional = 7
};
```

E.18 GOLF_MemberStatusEnum.mof

```
// ==================================================================
//  GOLF_MemberStatusEnum
// ==================================================================
enumeration GOLF_MemberStatusEnum : GOLF_ProfessionalStatusEnum
{
    Basic = 0,
    Extended = 1,
    VP = 2,
};
```

E.19 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 = "Maine",
   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"
};
```

E.20 JohnDoe.mof

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

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

value 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 = value of GOLF_Address
      {
         State = IL;
         City = "Oak Park";
         Street "Oak Park Av.";
         StreetNo = "1177;
         ApartmentNo = "3B";
      };
   MemberPhoneNo = $JohnDoesPhoneNo;
   MemberEmailAddress = "JonDoe@hotmail.com";
};
```

ANNEX F
(informative)

Change log

In earlier versions of CIM the MOF specification was part of the DSP0004. See ANNEX I in DSP0004 for the change log of the CIM specification.

Version	Date	Description
3.0.0	2012-12-13	DMTF Standard

Bibliography

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

W3C, XML Schema, *Part 2: Datatypes (Second Edition), W3C Recommendation 28 October 2004*
http://www.w3.org/TR/xmlschema-2/