An Explanation of the Architecture of the MMS Standard

written by:

Herbert Falk
SISCO

Jeffrey Robbins
Cycle Software, Inc.

Reviewing the Manufacturing Message Specification (MMS ISO/IEC-9506) can be overwhelming. The amount of information, and the ISO format, often cause the uninitiated to overlook the underlying rationale and design methodology. Losing the forest for the trees, most reviewers tend to concentrate on the actual details of the specification.

The creators of the standard spent a great deal of time attempting to format the document so that MMS implementors could easily find the details. However, in so doing, the overview and concepts that led to the particular format and services has been glossed over. This document will attempt to document the architecture of MMS in terms of its general model, objects, methods, and services.

Architecture

The MMS specification assumes an inherent understanding of the scope of the problem space that is being addressed by MMS. In general, the solution (MMS) provides for peer-to-peer "real-time" communications over a network. During the design of MMS, an initial attempt to standardize "real-world" devices (e.g. Programmable Logic Controllers, Numerical Controllers, and Robots) was attempted. The "real-world" operational characteristics of these various devices were so different that consensus on a single "real-world" device model could not be reached. The lack of consensus, coupled with the desire to allow MMS to be applied to solve problems areas, forced the creators of MMS to work on standardizing observable network behavior only.

The decision to standardize behavior, without direct correlation to "real-world" devices, caused the creation of a "virtual" model. The underlying assumptions of the MMS model, and its objects, are that they exist only when network communications are operating within a MMS Context.

The Virtual Manufacturing Device (VMD) is the MMS object which has at least one network-visible address. The addressing allows for an MMS Context to be negotiated between two peer applications. Once the context is established, the standard specifies the details of the MMS objects, attributes, hierarchy, and methods for the objects.

The following table details the MMS objects and their intended use:

MMS Model Object	Description
Context	The context object represents the attributes that are exchanged so that the MMS behavior is known to both cooperating applications prior to attempting to use other MMS services.
Virtual Manufacturing Device	The VMD itself can be viewed as the object in which all other MMS objects are contained. It has attributes that reflect general capabilities and a general set of methods that are inherited by all other MMS objects.
Named Variables	This class of object is, in general, used for "real-time" data exchange. Its intended use is for data monitoring, non-historic data reporting, and allowing data to be reported in an unsolicited fashion.
Named Variable List	This class of object is used to aggregate, into a list, other variable objects. It differs from the Named Variable object in that each element in the list (when accessed) returns its own success or failure.
ScatteredAccess	This class of object is, in general, used for "real-time" data exchange. It differs in capability from the Named and Named Variable List objects through its external behavior. The differentiating behavior is its ability to aggregate other variable objects and have the external appearance of creating a single coherent variable. The intended use of this object is to allow non-MMS variables to be

MMS Model Object	Description
	aggregated into complex MMS objects allowing for the migration of legacy protocols.
Named Type	This class of object is used to create a dictionary of well known, and re-usable, data type definitions which allow complex variables to be created. It allows the consistent definition of data representation and the associated range of values to be defined.
Semaphore	This class of object is intended to be used for the resolution of object/resource contention. The characteristics of this object were developed with the UNIX semaphore/token model as the basis.
Event Condition	This class of object is used to allow network applications to be able to determine the state of a condition (Active, Inactive, or Disabled). The specification does not specify the local processing required to transition the state of the object, but how to use the object to trigger network activity based upon the state transitions. The combination of EventCondition, EventAction, and EventEnrollment object use is intended for the construction of dynamic report by exception network scenarios.
Event Action	Instances of this class of object allows for a set of potential network actions to be defined. These actions are then linked to a particular EventCondition transition through the use of an EventEnrollment object. The combination of EventCondition, EventAction, and EventEnrollment object use is intended for the construction of dynamic report by exception network scenarios.
Event Enrollment	Instances of this class allow a network application to define that a particular EventAction object be performed upon a given EventCondition state transition. Further , it allows the specification of which network application should be notified of the occurrence of the state transition. The combination of EventCondition, EventAction, and EventEnrollment object use is intended for the construction of dynamic report by exception network scenarios.
Journal	This class of object is used for the exchange of historic or archived information. The object allows for data and network (MMS) transactions to be written into a “log” type of format that allows for the creation of a Sequence of Events (SOE) recording function. Likewise, the “logged” information can be retrieved so that applications can regenerate information whose time sequencing is important.
Domain	This class of object has been left intentionally vague within the MMS specification. The standard states that domains represent resources. Therefore, the first question is what is the definition of a “resource”. The intended definition of resource was any aggregation of objects, data, etc... required to perform a single function. It can contain execution instructions, MMS objects, and other information. In general, the concept was borrowed from the process control industries where batch processing needed to be facilitated. This object allows executive code (programs) and configuration settings to be uploaded/downloaded over the network.
Program Invocations	The behavior of this object was borrowed from the concept of a UNIX Execution Thread. It is the object that is used to start and stop remote application processes.

MMS Model Object	Description
Operator Station	This object facilitates a poor-man's Telnet exchange of information. The use of this object is restricted to the exchange of simple text messages for human operators.
File	This class of object is to be used to transfer binary file information. It does not provide record access but transfers files in their entirety.

Each of the fifteen (15) MMS model objects, except for Operator Station, have five methods that are in common. These are:

MMS General Methods	Description
Get	This method is used to obtain the value or contents of a specified object.
Set	This method is used to write/put value or contents into a specified object.
QueryAttributes	This method is used to obtain structure or capability information of a specified object.
Create	This method allows objects of particular classes to be instantiated (come into existence).
Delete	This method allows instantiated objects to be destroyed (removed from existence).

MMS provides a complete *architecture* for solving distributed automation problems. The MMS architecture is based on a *model* that has specific applicability to real world problems by providing meaningful *objects*. These meaningful objects provide the semantic capability of MMS and correspond to specific functions of real Intelligent Electronic Devices (IEDs) and applications. As we have seen, each of these MMS model objects inherits the basic five *methods*. The methods are implemented via specific *services* which carry the actual *parameters* required for the specific object.

This specification in terms of services can be mapped onto current object-oriented methodology by viewing each object-specific service as an overload on the inherited method -- however, the devil is in the details of what you mean by "overload". We must not overlook the important differences between doing a Get on a variable object and doing a Get on a journal object. Each MMS model object is different enough to warrant being distinct from the other objects. These differences are captured in the details of the services and parameters that implement each method. We can examine this in depth by taking two model objects, a variable list and a journal, and compare the MMS specification of the Get method.

In the case of a MMS variable list, the Get method maps onto the Read service. Let us examine in depth one phase of the Read service, the Read-Request:

```
Read-Request ::= SEQUENCE
    {
    specificationWithResult        [0] IMPLICIT BOOLEAN DEFAULT FALSE,
    variableAccessSpecification    [1] VariableAccessSpecification
    }
```

The Read-Request lets us Get one or more variables from a VMD. It starts off with a Boolean that lets us choose how we want the VMD to return the result, with the requested variable specification or without it. This is tagged as IMPLICIT and also DEFAULTs to FALSE -- we don't normally need to be reminded about what we requested when we get a response!

The Read-Request then refers to a VariableAccessSpecification which looks like this:

```
VariableAccessSpecification ::= CHOICE
    {
    listOfVariable        [0] IMPLICIT SEQUENCE OF SEQUENCE
        {
        variableSpecification   VariableSpecification,
        alternateAccess         [5] IMPLICIT AlternateAccess OPTIONAL
        },
    variableListName      [1] ObjectName
    }
```

Here we can see that we have a choice between an enumerated list of variables (and optional components called *AlternateAccess*) or a named variable list object. This is another key detail of the Get method which MMS builds right into the model object. It is worth going in one more level of detail to see what a VariableSpecification looks like in case we want to enumerate our variables:

```
VariableSpecification ::= CHOICE
    {
    name                        [0] ObjectName,
    address                     [1] Address,
    variableDescription         [2] IMPLICIT SEQUENCE
        {
        address              Address,
        typeSpecification    TypeSpecification
        },
    scatteredAccessDescription  [3] IMPLICIT ScatteredAccessDescription,
    invalidated                 [4] IMPLICIT NULL
    }
```

Here we can see that we also have choice among names, addresses, descriptions of both type and address and also scattered descriptions. The full extent of the Get method for a variable list is revealed if we examine what the *TypeSpecification* from this looks like:

```
TypeSpecification ::= CHOICE
    {
    typeName            [0] ObjectName,
    array               [1] IMPLICIT SEQUENCE
        {
        packed              [0] IMPLICIT BOOLEAN DEFAULT FALSE,
        numberOfElements    1] IMPLICIT Unsigned32,
        elementType         [2] TypeSpecification,
        },
    structure               [2] IMPLICIT SEQUENCE
        {
        packed              [0] IMPLICIT BOOLEAN DEFAULT FALSE,
        components          [1] IMPLICIT SEQUENCE OF SEQUENCE
            {
            componentName       [0] IMPLICIT Identifier OPTIONAL,
            componentType        [1] TypeSpecification
            }
        },
-- Simple Type --------------------- Size ---------
    boolean             [3] IMPLICIT NULL,
    bit-string          [4] IMPLICIT Integer32,
    integer             [5] IMPLICIT Unsigned8,
    unsigned            [6] IMPLICIT Unsigned8,
    floating-point      [7] IMPLICIT SEQUENCE {
        format-width        Unsigned8,
        exponent-width      Unsigned8
        },
    real                [8] IMPLICIT SEQUENCE {
        base                [0] IMPLICIT INTEGER (2|10),
        exponent            [1] IMPLICIT INTEGER OPTIONAL,
        mantissa            [2] IMPLICIT INTEGER OPTIONAL
        },
    octet-string        [9] IMPLICIT Integer32,
    visible-string      [10] IMPLICIT Integer32,
    generalized-time    [11] IMPLICIT NULL,
    binary-time         [12] IMPLICIT BOOLEAN,
    bcd                 [13] IMPLICIT Unsigned8,
    objId               [15] IMPLICIT NULL
    }
```

What we can see here is that when we want to do the Get method on a variable list, MMS lets us drill all the way down to a detailed specification of recursive data structures which allows access to entire pieces of data, data scattered over multiple memory locations, and specific pieces of data contained within a IED. None of this depends on application logic in either the host application or the IED protocol implementation; it is built in to the MMS software!

Now let us compare this Get to a Get of a Journal object. MMS has a ReadJournal service; we will focus on the request phase:

```
ReadJournal-Request ::= SEQUENCE
    {
    journalName                 [0] ObjectName,
    rangeStartSpecification     [1] CHOICE
        {
        startingTime                [0] IMPLICIT TimeOfDay,
        startingEntry               [1] IMPLICIT OCTET STRING
        } OPTIONAL,
    rangeStopSpecification      [2] CHOICE
        {
        endingTime                  [0] IMPLICIT TimeOfDay,
        numberOfEntries             [1] IMPLICIT Integer32
        } OPTIONAL,
    listOfVariables     [4] IMPLICIT SEQUENCE OF VisibleString OPTIONAL,
    entryToStartAfter           [5] IMPLICIT SEQUENCE
        {
        timeSpecification           [0] IMPLICIT TimeOfDay,
        entrySpecification          [1] IMPLICIT OCTET STRING
        }
    }
```

We can see similarities and differences between this request and the Read-Request. Item [4] of this request is a list of variables we want journal entries about. This seems similar to the Read-Request above, but the list here is a much simpler sequence of strings. We get none of the flexibility we have in the Read-Request to access by name or by description. Here the presumption is if you want to key into a journal on a variable, you had better give the variable a name. In addition to this parameter, the Read-Journal request lets us access a set of entries falling within a range of time. We can also specify the number of entries we want.

So for these two MMS model objects -- variable list and journals -- we can see two strikingly different sets of parameters on the MMS services that implement the Get method. The reader might ask at this juncture: “OK, I understand the differences, but are the differences a good thing?” Since the differences are a function of specificity, the question could be recast as a critique of MMS as either being too specific or not specific enough.

The MMS standard is specific enough to capture a common application protocol that is applicable to a wide range of applications and equipment and to allow implementors to build real problem solving tools. Without this specificity, the protocol becomes empty syntax lacking the semantic weight to carry a significant share of the application burden. Interoperability requires more than syntax; it demands shared meaning so that real applications can connect to real equipment to solve real problems. However, there is flexibility allowed within the selection of objects and services that are used to solve a particular problem.

This flexibility leads to the question: Is MMS not specific enough? In a sense, yes. The ambiguity associated with the application object definitions is intentional. The creators/editors of the MMS standard realized that expertise required for object and service standardization would need to be supplied from experts within a given problem domain. Therefore, activities similar to the Electric Power Research Institute’s (EPRI’s) MMS Forum are required. This type of meeting represents a gathering of experts within an application area, an industry, or a product category to define specific object types (and possibly additional parameters) to target the standardized network behavior of MMS into its problem domain. This allows interoperable mapping of the virtual model onto real equipment. MMS combined with standard object definitions will allow real business needs to be met in an interoperable and extensible manner.

In conclusion, we have seen that by mapping from method to service/parameters in the ISO/IEC 9506-1 specification, MMS has lent itself to interoperable semantics. In addition, by mapping these services/parameters into concrete syntax in ISO/IEC 9506-2, MMS has created an interoperable syntax. By containing both syntax and semantics, the MMS architecture thus provides a completely self-contained methodology for object-oriented interoperability over the wire.

Appendix 1 - Table of Methods vs. MMS Services

	Methods				
Objects	**Get**	**Set**	**Query**	**Create**	**Delete**
Context		Cancel Reject		Initiate	Conclude /Abort
VMD		Rename	Status/Identify/GetCapabilityList GetNameListUnsolicitedStatus		
Named Variables			GetVariableAccessAttributes	DefineNamedVariable	DeleteVariableAccess
ScatteredAccess			GetScatteredAccessAttributes	DefineScatteredAccess	DeleteVariableAccess
Named Variable List	Read InformationReport	Write	GetNamedVariableListAttributes	DefineNamedVariableList	DeleteNamedVariableList
Named Type			GetNamedTypeAttributes	DefineNamedType	DeleteNamedType
Operator Station	Input	Output			
Semaphore	RelinquishControl	TakeControl	ReportSemaphoreStatus ReportPoolSemaphoreStatus ReportSemaphoreEntryStatus AttachToSemaphore	DefineSemaphore	DeleteSemaphore
Domain	InitiateUploadSequenceUpload Segment TerminateUploadSequence RequestDomainUpload	InitiateDownloadSequence DownloadSegmentTerminateDo wnloadSequence RequestDomainDownload	GetDomainAttributes	LoadDomainContent StoreDomainContent	DeleteDomain
Program Invocations		Start/ Stop/ Resume/ Reset/ Kill	GetProgramInvocationAttributes	CreateProgramInvocation	DeleteProgramInvocation
Event Condition	ReportEventConditionStatus	AlterEventConditionMonitoring TriggerEvent	GetEventConditionAttributes	DefineEventCondition	DeleteEventCondition
Event Action	ReportEventActionStatus		GetEventActionAttributes	DefineEventAction	DeleteEventAction
Event Enrollment	ReportEventEnrollmentStatus EventNotification GetAlarmEnrollmentSummary GetAlarmSummary	AlterEventEnrollment AcknowledgeEventNotification AttachToEventCondition	GetEventEnrollmentAttributes	DefineEventEnrollment	DeleteEventEnrollment
Journal	ReadJournal	InitializeJournal WriteJournal	ReportJournalStatus	CreateJournal	DeleteJournal
File	FileRead	ObtainFile FileRename	FileDirectory	FileOpen /FileClose	FileDelete

Appendix 2 - Object Schema of MMS

Object Schema of MMS

Most MMS model objects have an architectural hierarchy defined. However, many of the objects also have a associated object SCOPE .

The object SCOPEs, as defined in the MMS specification are:

1. VMD Wide - Objects of this scope are global and are accessible through any association.

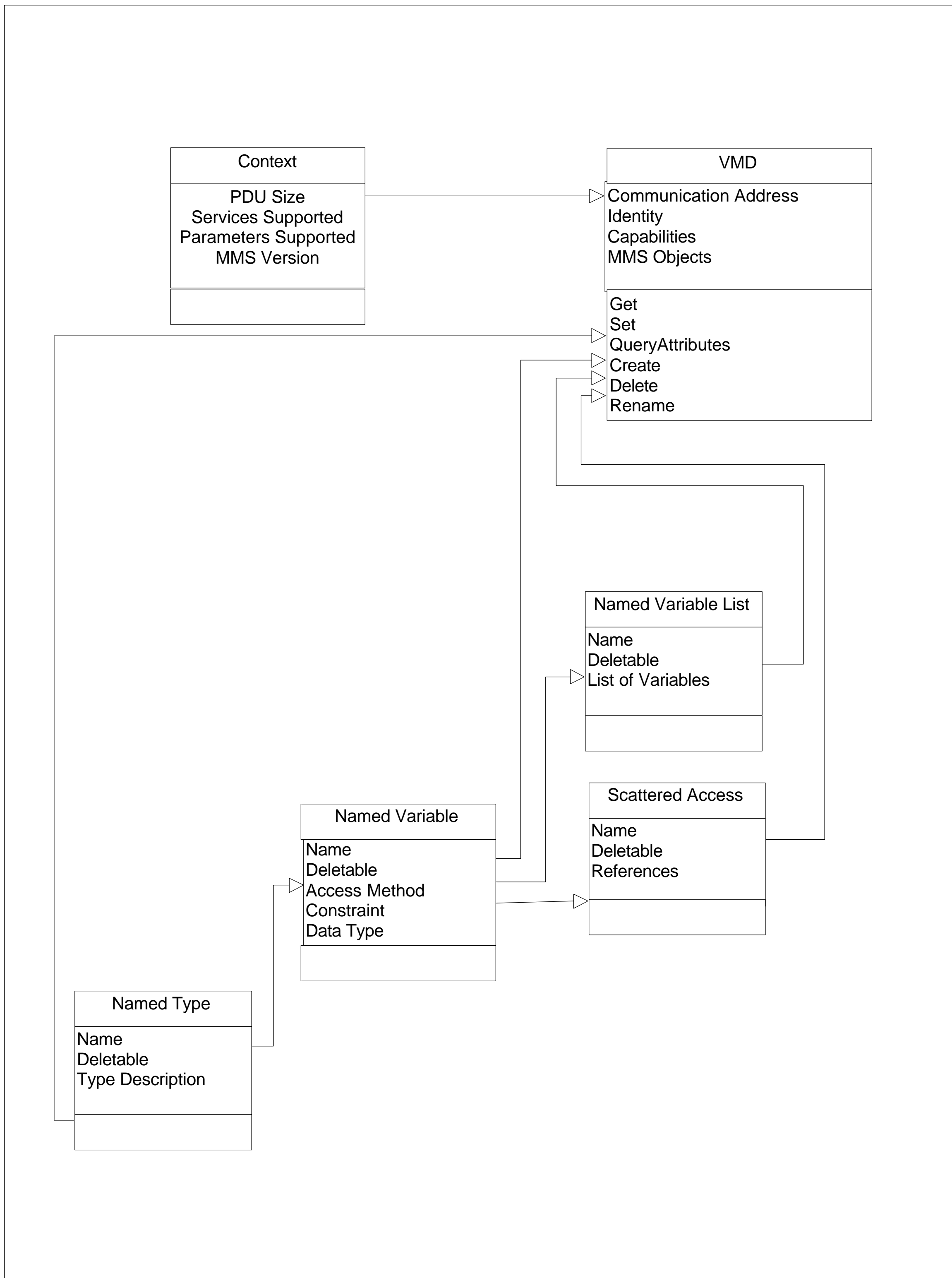
2. Domain Specific - Objects of this scope are bound to a particular MMS Domain Object. The objects are accessible through any association.

3. Association Specific - Objects of this scope are associated with a single association and are not available through any other association.

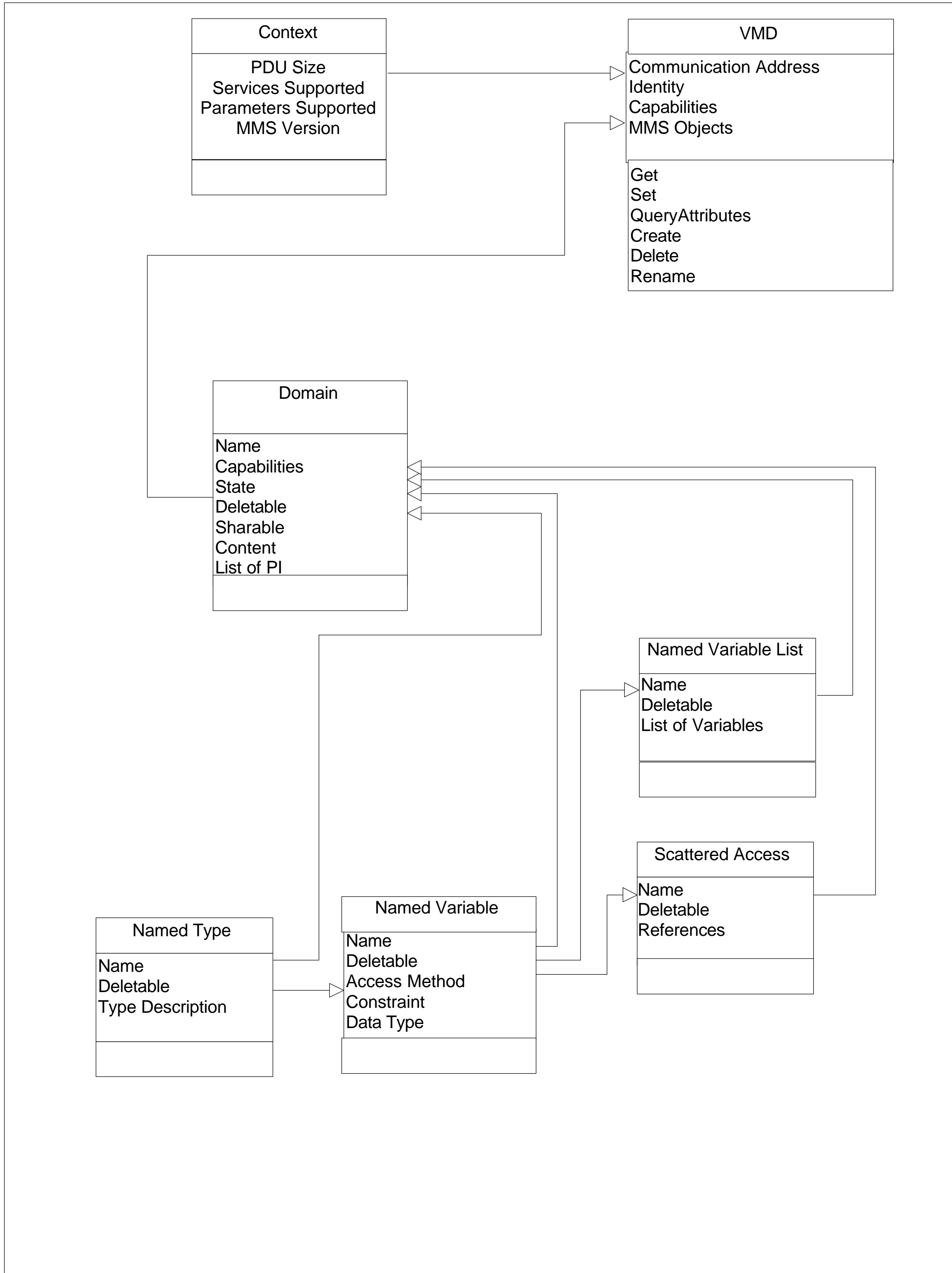
The following table shows the MMS model objects with their allowed SCOPEs.

	Object Scope		
Objects	VMD	Domain	Association-Specific
Named Variable	x	x	x
ScatteredAccess	x	x	x
Named Variable List	x	x	x
Named Type	x	x	x
Semaphore	x	x	
Event Condition	x	x	x
Event Action	x	x	x
Event Enrollment	x	x	x
Journal	x		x
Domain	x		
Program Invocation	x		
Operator Station	x		

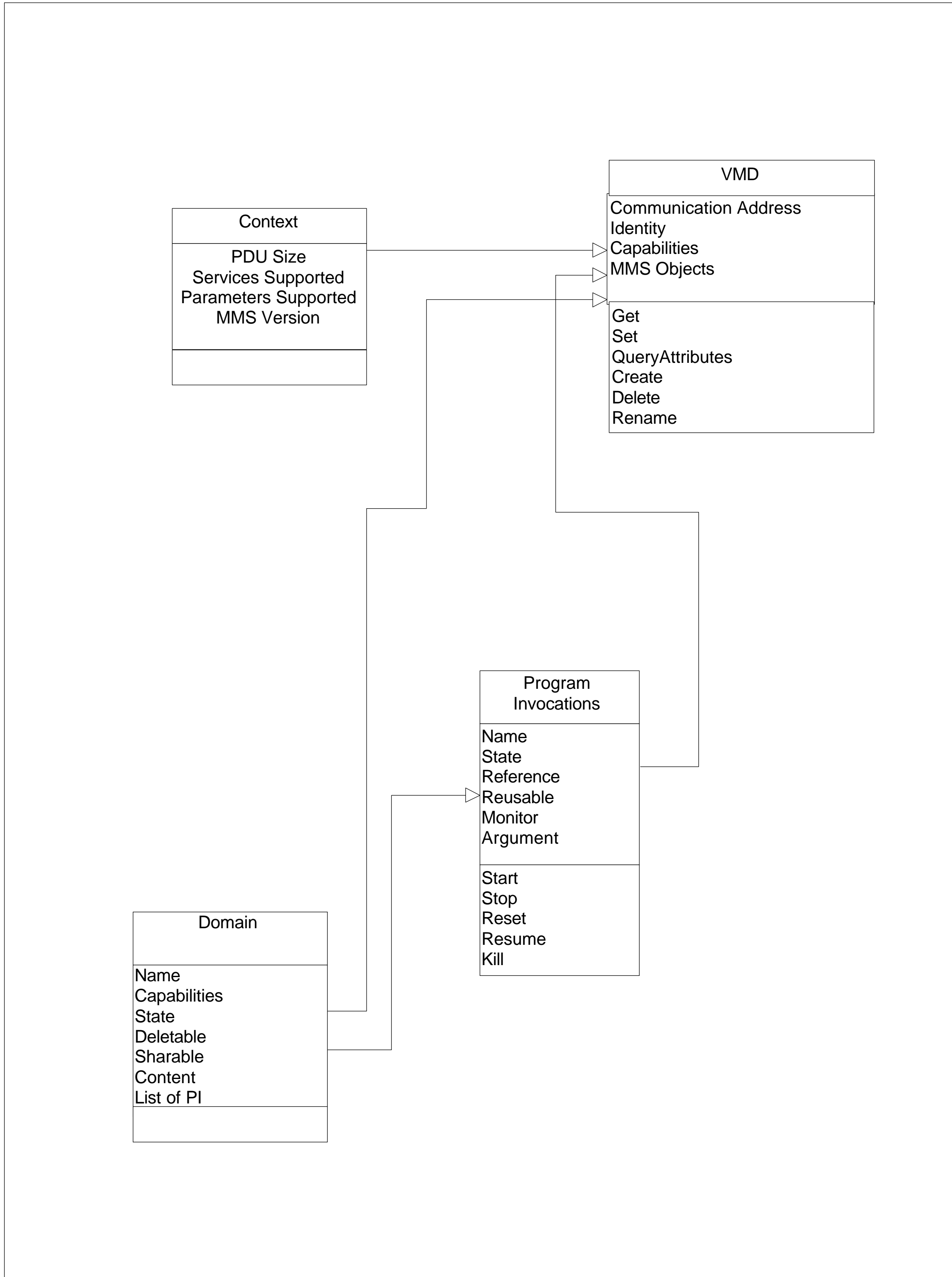
The following diagrams represent a simplified view of the object hierarchy and methods (Association-Specific relationships are not shown).



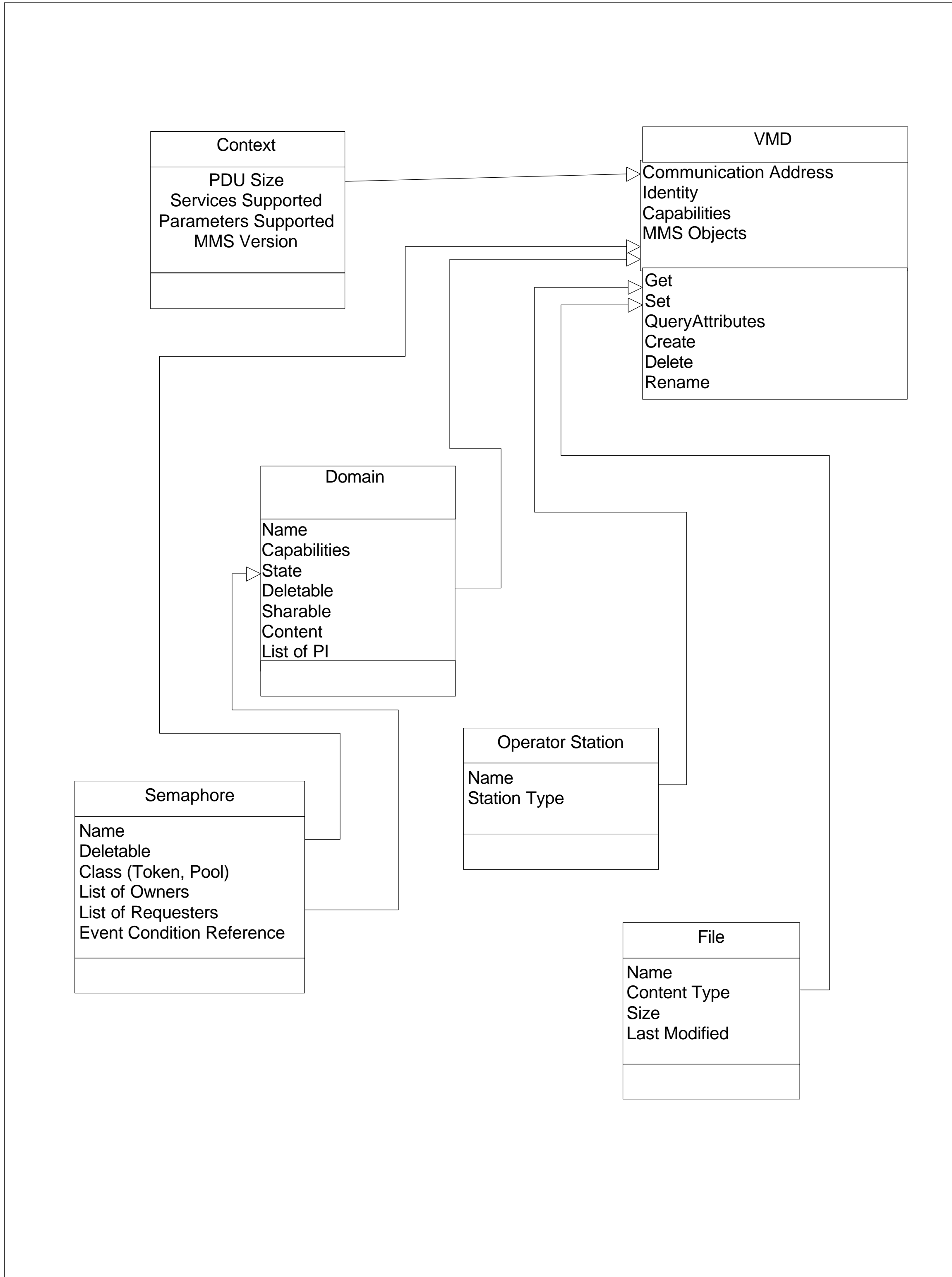
VMD Scope Variables



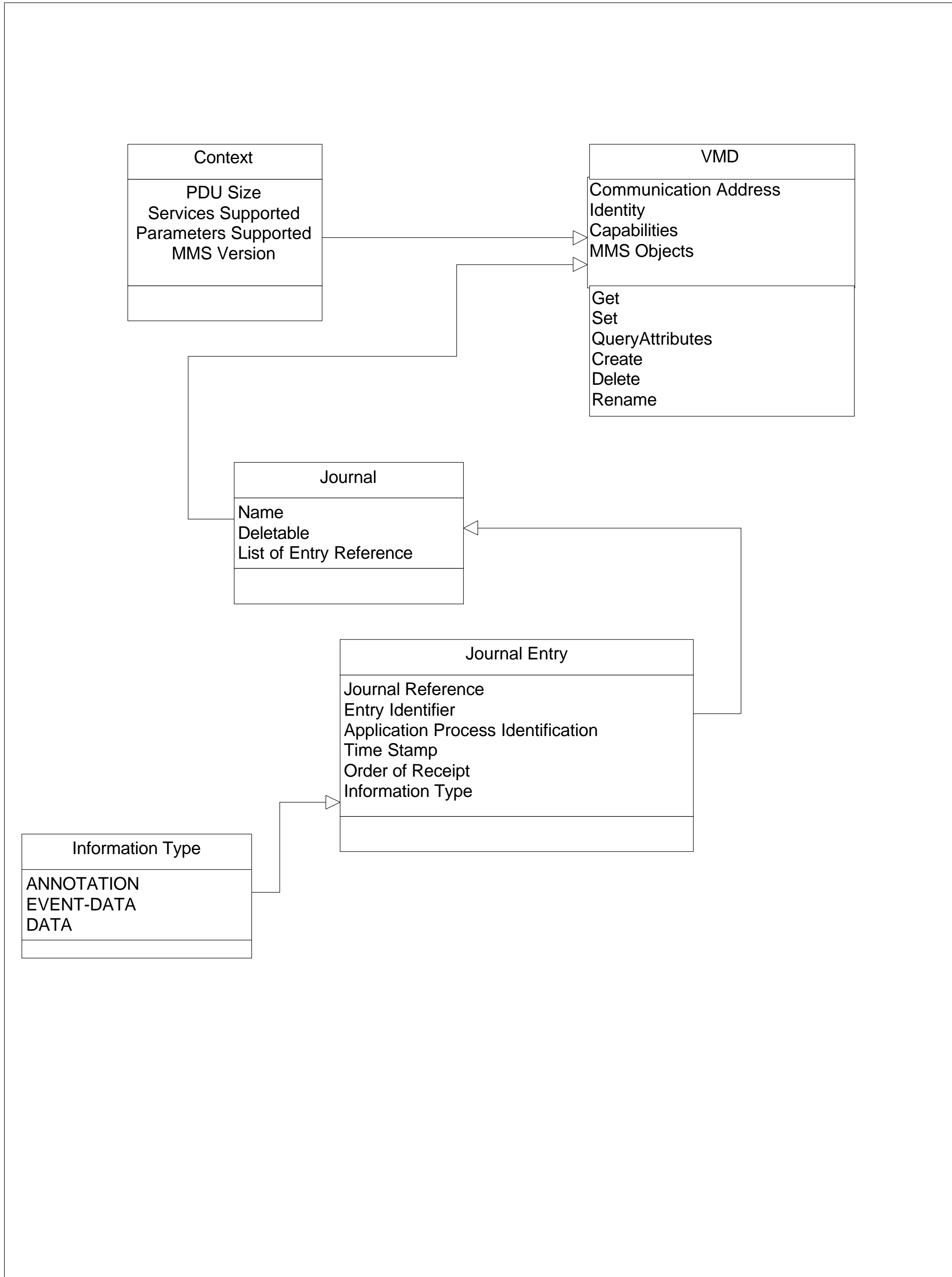
Domain Specific Variables



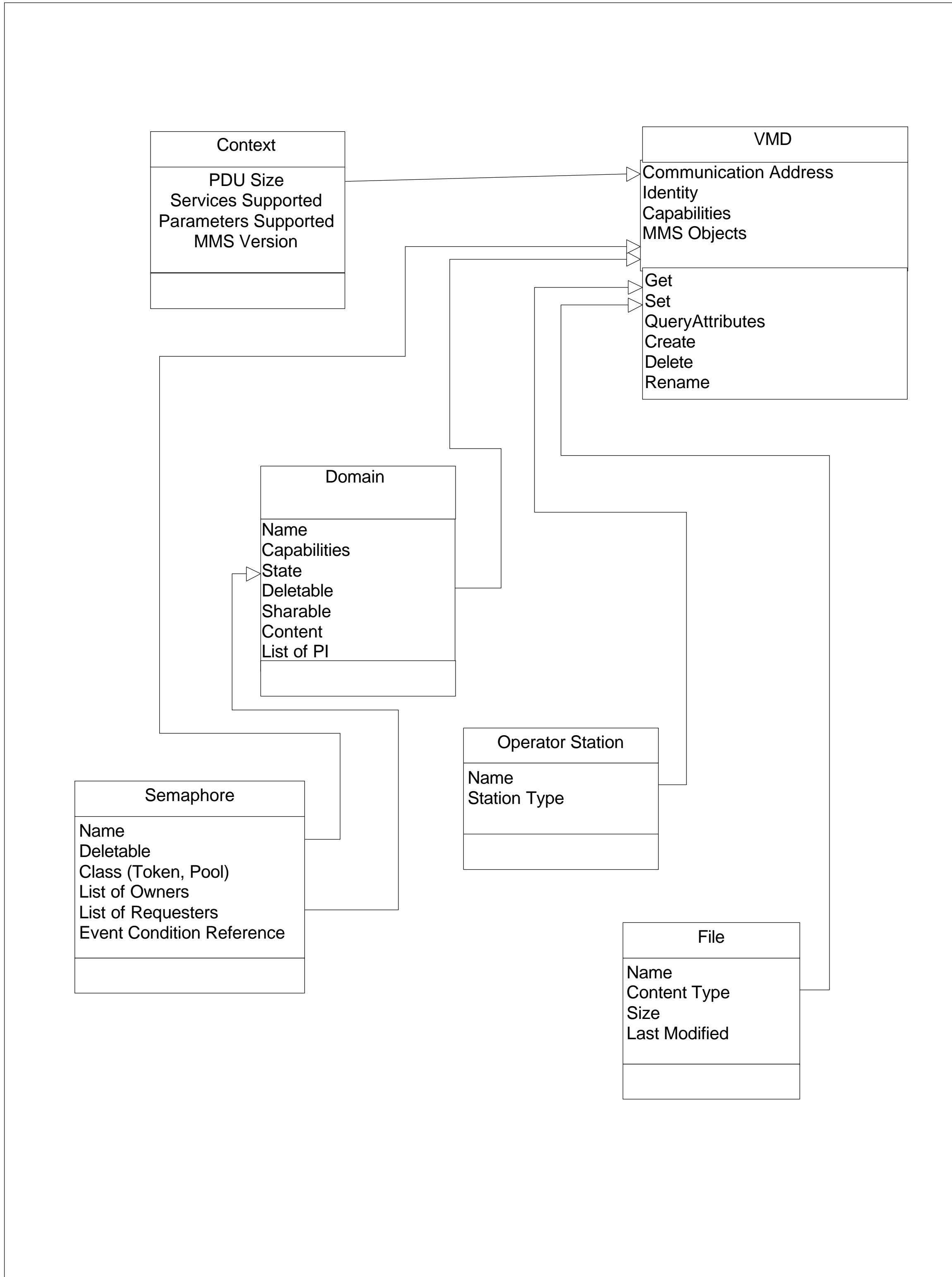
Domains and Program Invocations



Semaphores and Files



Journals



Event Condition, Action, and Enrollment Objects