miUML Metamodel Class Descriptions
Domain Subsystem

Leon Starr

Sunday, July 24, 2011

mint.miUMLmeta.td.2

Version 2.1.2

Table of Contents

Introduction

This metamodel captures the semantics of Executable UML as described in the book Executable UML: A Foundation for Model Driven Architectures. This document explains how the Class Model Metamodel supports these rules. Justification and explanation of Executable UML rules themselves are beyond the scope of this metamodel.

These descriptions are part of the Executable UML Class Model Metamodel Class Model. Some of the Classes in this model are well known entities defined in [Steve REF]. The beginning of each such description starts with the phrase "Defined in Executable UML". These descriptions do not attempt to duplicate those definitions. But for convenience, concise statements are included here. For further details, please read these books. All of the other classes were invented for the purposes of this metamodel. Full descriptions are included.

Key to Styles and Conventions

COLOR is used to convey information in this document. If you have the ability to print or display this document in color, it is highly recommended. Here are a few notes on the font, color and other styles used in class model descriptions.

Naming classes, types and subsystems

Modeled elements such as classes, types and subsystems are written with initial caps. The text Air Traffic Controller, for example, would probably be a class. If you see the text Application Domain, you know it is a modeled element, most likely a domain.

Naming attributes

Attributes are named with an initial cap followed by lower case, Time_logged_in, for example.

Instance values

Instance data, such as attribute values, are either in quote marks as in "Gwen" and "Owen" or represented in the following color/font: Gwen and Owen.

Whitespace

When programming, the squashedNamingStyle is optimal for easy typing and is adequately readable for short names. That's fine for programming, but analysis is all about easy readability and descriptive, sometimes verbose names. Programming style is retained for method names, but everywhere else whitespace or underscores are employed. For example: Air Traffic Controller, Time logged in or On_Duty_Controller.Time_logged_in.

hardcoreJavaProgrammersMayDisputeThisPointAndArgueThatWhiteSpacesAreEntirelyUnnecessaryForImprovingReadabilityButIdisagree.

Attribute heading color

Attribute headings are colored according to the role the attribute plays in its class. Referential attributes are brown, **Logged in controller**, for example. Naming attributes are blue, **Number**, for example. Finally, descriptive attributes are red, **Time logged in**, for example. Derived attributes are purple **\Volume**, for example.(One of the nice things about color is that it helps eliminate the need for underscores).

Referential renaming

A referential attribute will often have a name that reflects its role rather than the name of the base reference. Station.Logged_in_controller may refer to On_Duty_Controller.ID, for example. Such renaming will be defined in the referential attribute's description.

MDA (Model Driven Architecture) Layers

The MDA defined layers are referenced from time to time. It is sometimes useful, especially in a metamodel, to distinguish the various meta layer names: M0, M1, M2 and M3. These are M0: data values ('N3295Q', 27L, 490.7 Liters, ...) M1: domain specific model elements (class 'Aircraft', attribute 'Altitude', relationships 'R2 - is piloted by'), M2: metamodel elements to define miUML (class: Class, class: Attribute, class: Relationship, attribute:'Rnum'...) and M3: meta-metamodel elements to define (ambitiously) any modeling language (class 'Model Node', class 'Structural Connection'). For our purposes, we will seldom refer to the M3 layer and M2 will be relevant only if the subject matter at hand is a metamodel.

Keep in mind that these layers may be interpreted relative to the subject matter at hand. If you are modeling an air traffic control system, 490.7 Liters is likely data in the M0 layer with the class 'Aircraft' at the M1 layer. 'Class' would be at M2 - the metamodel layer.

But if the subject matter is Executable UML, as is the case in the miUML metamodels, both the class 'Aircraft' and 490.7 Liters could be compressed into the M0 layer. ('Aircraft' is data populating the Class and Attribute subsystem and 490.7 Liters populated into the Population subsystem). At the M1 layer we would possibly have 'Class' and 'Value'. And, since the metamodel will populate itself, as we bootstrap into code generation, the M2 layer is also 'Class' and 'Value'.

In other words, we can create a data base schema from the miUML metamodel and then insert the metamodel itself into that schema. So if we define a table 'Class' based on our metamodel in the data base, we could then insert metamodel classes 'Class', 'Attribute', etc. into that table. Thus we see 'Class' both at the M0 (inserted data) and M1 (modeled) and M2 (metamodel) layers!

Local Data Types

The following data types are used by attributes in this subsystem.

Name

This is a string of text that can be long enough to be readable and descriptive. A handful of words is okay, a full tweet is probably too much. Since the exact value may change to suit a particular platform, it is represented symbolically as "LONG_TEXT".

Domain: String [LONG_TEXT]

Nominal

This is a whole number used purely for naming with no ordinal or computational properties.

Identifiers used only for referential constraints may be stripped out during implementation (assuming the constraints are still enforced). During simulation and debugging, however, these can be nice to have around for easy interpretation of runtime data sets without the need for elaborate debug environments. "File-6:Drawer-2:Cabinet-"Accounting" is easier for a human to read than a bunch of pointer addresses.

Domain: The set of positive integers {1, 2, 3, …}.

Description

This is an arbitrarily long amount of text. The only length limit is determined by the platform.

References

[MB] Executable UML: A Foundation for Model-Driven Architecture, Stephen J. Mellor, Marc J. Balcer, Addison-Wesley, 2002, ISBN 0-201-74804-5

[DATE1] An Introduction to Database Systems, 8th Edition, C.J. Date, Addison-Wesley, 2004, ISBN 0-321-19784-4

[LSART] How to Build Articulate UML Class Models, Leon Starr, Model Integration, LLC, Google Knol, http://knol.google.com/k/how-to-build-articulate-uml-class-models

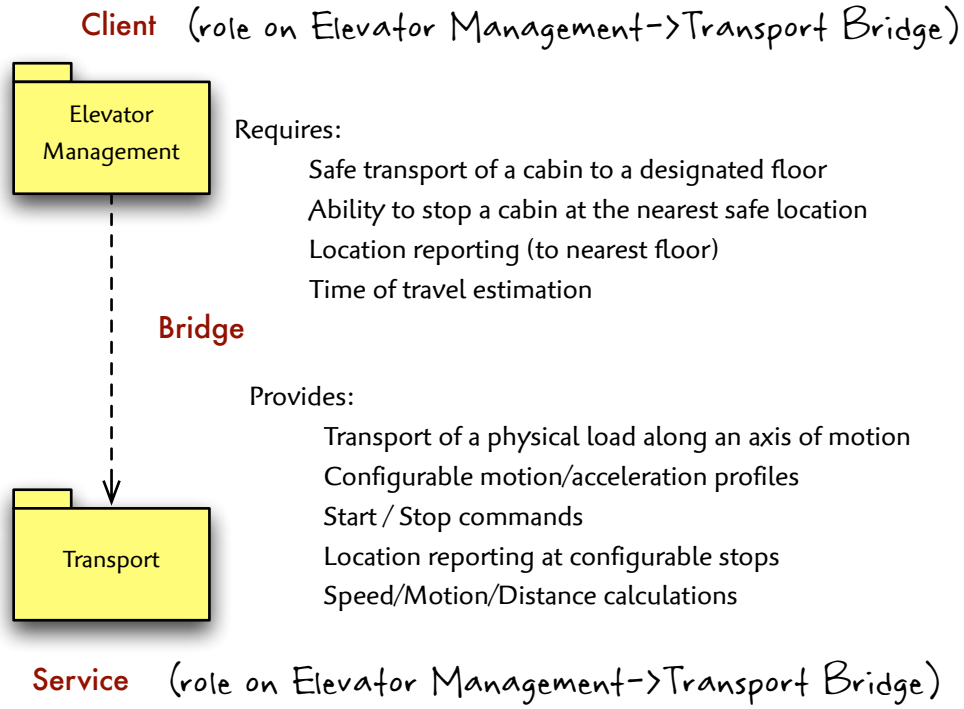
[OOA96] Shlaer-Mellor Method: The OOA96 Report v1.0, Sally Shlaer, Neil Lang, Project Technology, Inc., 1996 (Available as a PDF download at www.modelint.com)

Bridge BRIDGE

"A Bridge is a layering dependency between Domains. One Domain makes the assumptions, and other Domains take those assumptions as requirements." [MB]

miUML uses the definition in Executable UML above.

There are a few ideas associated with the Bridge concept. From an analysis perspective a dependency is established, in the context of a project, such that a Domain playing the role of client requires the capabilities of a Domain playing the role of service. In fact, a Domain may play a different role on each Bridge, so the same Domain may be a client on one Bridge and a service on another.



The capabilities supplied by the service may be bound at any time such as compile, initialization or runtime. It is a common mistake to think only in terms of runtime binding.

A runtime binding might be implemented by remote procedure calls (assuming the two domains are implemented as separate tasks which is only one possible deployment). Other dynamic interactions are possible such as function calls, messages, events, etc depending on the platform.

An initialization binding might involve the population and configuration of instance data in the service based on M0 (objects/links/attribute values) and M1 (classes, attributes, associations, etc) from the client. During runtime, this data may be consulted without the need, or a minimal need, to interact with the client.

A compilation binding would be similar to initialization except that the population/configuration would result in the generation of header files and other code to implement the service. Again, the service would run without the need, or only a minimal need, to interact with the client.

To get any of these three scenarios to work, however, it will be necessary for the analyst/modeler to define mappings between elements of the client Domain and elements of the service Domain. Ideally, these mappings should be among metamodel elements. In practice, a mechanism for specifying these mappings is imperfectly provided and varies from one platform architecture to another, depending on the nature of both the application and platform. This mechanism usually consists of some combination of marking models and explicit bridging actions which are encoded directly in Executable UML action language. That is because a universal, platform independent metamodel mapping system is yet to be defined. This is a future, and extremely important, miUML project. For now, only the overall dependency from one Domain to another is modeled without any exposure of how the Bridge mechanics are defined.

Attributes

Client

Domain.Name — The Domain making assumptions and supplying requirements.

Service - constrained

Domain.Name — The Domain fulfilling the assumptions and satisfying the requirements established by the client. Constrained so that a service Domain may not also be its client.

Identifiers

1> Client + Service

Standard identifier composition rules for a Mx:Mx associative class. There can be only one Bridge between the same two Domains.

Relationships

R7
Domain MAKES ASSUMPTIONS ON *zero, one or many* Domain
Domain SATISFIES ASSUMPTIONS OF *zero, one or many* Domain

In a simple system, there is only one all encompassing Domain. A more realistic project will be divided into multiple Domains with a primary application Domain defining the overall purpose of the entire software system. This top-level Domain will not serve any clients and will rely upon one or more Domains providing services.

A Domain providing a service may satisfy multiple client Domains. In fact, any well abstracted Domain should be configurable for multiple client Domains. The relationship under consideration here expresses the client-service role within a single project (domain chart/model). While

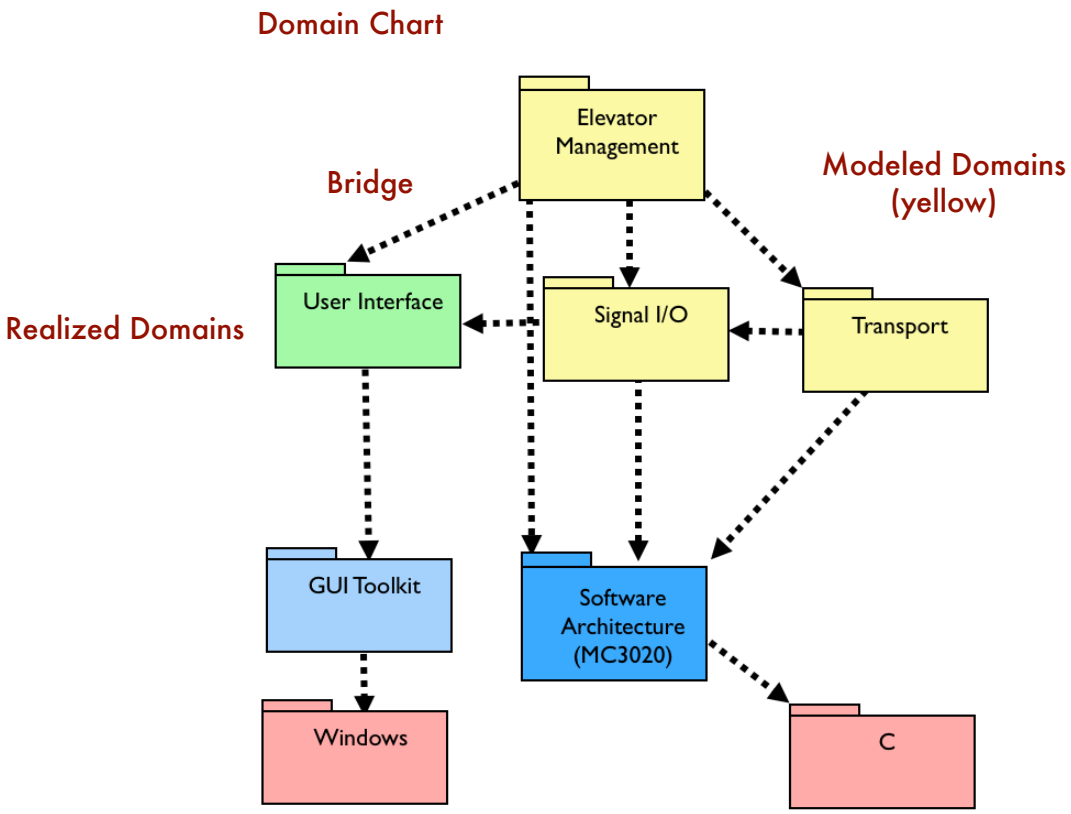
a Domain may potentially be useful in many different projects, the question is, in a single project, may a Domain satisfy the assumptions of more than one client Domain? Here the answer is a definite 'Yes'. For example, consider a logging domain that gathers data from multiple client Domains in the same project.

Domain DOM

"A Domain is an autonomous, real, hypothetical, or other abstract world inhabited by a set of conceptual entities that behave according to characteristic rules and policies." [MB]

miUML uses the definition in Executable UML above.

A Domain is represented on a miUML domain chart as a UML package which contains a complete executable system. This may be a miUML model set, handwritten code, legacy code or any other executable software specification.



Each Domain is a black box with respect to any other Domain. This means that no component in one Domain should know about the structure of a component in another Domain. Furthermore, each Class* should appear only once in a system and in only one Domain. In the Elevator Application, the cabin class appears only in the Elevator Management Domain. Both the UI Domain and the Transport Domains appear to need the cabin concept, but this is not really the case. The UI provides an animated icon which can be configured to represent cabin instances. The Transport Domain provides a load which can be configured to represent a cabin. Each class, cabin, load and animated icon has a distinct definition and set of rules and constraints within each Domain. Mappings are provided on each Bridge to knit the concepts together during implementation.

All Domains and interconnecting Bridges used to construct a single software system or project are typically drawn on a miUML domain chart. The same Domain may be reused in multiple systems / projects. In fact, this is an intended by product of domain based system partitioning.

* A Realized Domain won't specify any miUML Classes, but the general notion of structures or entities still applies. A realized GUI package, for example, probably won't supply an elevator cabin entity unless it is extraordinarily application specific.

Attributes

Name

A descriptive name that would appear on a domain chart, Elevator Application or Graphical User Interface, for example.

Type: Name

Alias

An abbreviated name used for labeling purposes, EVAPP or GUI, for example.

Type: Short Name

Identifiers

1> Name

No two Domains may have the same name.

2> Alias

No two Domains may have the same Alias. (An Alias must be specified for each Domain).

Relationships

None.

Domain Build Specification DBSPEC

A few values must be supplied when a new Domain is defined. To make this easier, a number of configurable default values may be established. For example, the name of the required initial Subsystem can be automatically set to something like Main. Alternatively, the name of the Domain itself may be reused as the name of the initial Subsystem.

This is a singleton class because it is a universal set of data and rules to be applied to any Domain.

Attributes

Name

The name of the singleton. The value can be something like the spec.

Type: Name

Domain name is default subsys name

If set to TRUE, the initial Subsystem will share the same name as its Domain. A newly created Domain called Application, for example, will have an initial Subsystem titled Application. In this case the Default_subsys_name attribute value will be ignored.

Type: boolean

Default subsys name

This is the name of the initial Subsystem for any newly created Domain. If you set it to 'Main', for example, and create a new Domain titled 'Application', its initial Subsystem will be titled 'Main'.

Type: Name

Default subsys range

The extent of the default number range for Classes (Cnum) and Relationships (Rnum) within a Subsystem is specified here. If the value is 100, the initial Subsystem will have a range from 1-99 with subsequent ranges numbered 100-199. The first range always has one less unit (99 in this case) since we must start at 1. If the user edits the initial range to 1-50, leaving the default at 100 and then creates a new Subsystem, the new Subsystem will start out with a range of 51-150. So if d is the default subsys range and s is the high value of the next lowest subsystem range, the formula for creating a subsystem numbering range is: [1, d | s+1, s+d]

Type: Posint

Default id name

An miUML Class must have at least one Identifier. Consequently, a newly created Class requires at least one single Identifier Attribute. Use this name for the initial name of that Attribute. If the value is Number, for example, and a Class named Dog is created, this class will be defined with a single attribute Number as its Identifier. The data type of this Attribute will be the specified default System Type.

Type: Name

Default id type

Common Type.Name — The Common Type (data type) applied to an automatically created default Identifier Attribute.

Identifiers

1> Name

Relationships

R2

Domain Build Specification SPECIFIES DEFAULT ID ATTRIBUTE WITH *exactly one* System Type

System Type IS USED BY DEFAULT ID ATTRIBUTE SPECIFIED IN *zero or one* Domain Build Specification

The default id attribute in a Domain Build Specification, like any Attribute, requires a Type to be complete. Typically, this will be something like 'Nominal', 'Integer', 'Arbitrary', 'Identifier', 'Name' or something else that would be useful for general identification purposes. 'Color' is probably not a good idea.

The Type must be a System Type since the Domain Build Specification applies outside the scope of any particular Domain. This is just the default initial Identifier Attribute type. The modeler can change an Attribute's Type to a Domain Type, if desired, after the Attribute and its Class has been created.

A Type may or may not be the one used as the default id attribute 's Type.

Element ELE

An Executable UML model component confined and managed within a single Domain is known as an Element. This abstraction makes it possible to establish unique identity for all components of an Executable UML model.

It might appear that every conceivable model component should, therefore, be classified as a subclass of Element, but this is unnecessary. The uniqueness of an Attribute component, for example, is scoped within a Class, and therefore constrained to be within a single Domain. Most other model components are similarly scoped. Notable exceptions are Relationship, Class, Type and Constrained Loop. So each kind of Element has its own independent identity, outside the purview of anything else, within a Domain.

An Element may not span from one Domain into another. The same Element may not be present in multiple Domains.

Although it is usually bad practice, a Class of the same name may appear in two different Domains. But each would be a distinct Element. Presumably, even though the Class names match, the meanings would differ significantly (Event in a Notification Domain vs. Event in a Music Concert application Domain, for example).

Attributes

Number

Each Element is numbered uniquely within a Domain.

Type: Nominal

Domain

Domain.Name — The Element lives entirely inside this Domain.

Identifiers

1> Number + Domain

Established by policy.

RELATIONSHIPS

R15

Element IS MODELED IN *exactly one* Domain
Domain DEFINES *zero, one or many* Element

By definition, an Element is modeled within a single Domain. The Aircraft Class, for example, would be a component of a Domain titled Air Traffic Control. Another Class, also called Aircraft might appear in a different Domain titled Travel Itinerary Management. The Air Traffic Control Aircraft Class is not the same as the Travel Itinerary Management Aircraft Class because they are distinct Elements.

In practice, an Element will be identified by Domain.Name + <local_id> where the <local_id> may be Class_name, Rnum, or whatever is appropriate for a given Element subclass.

A Domain might be newly created and not include any Elements at all (though it will encompass at least one Subsystem).

R16

Element IS A Spanning or Subsystem Element

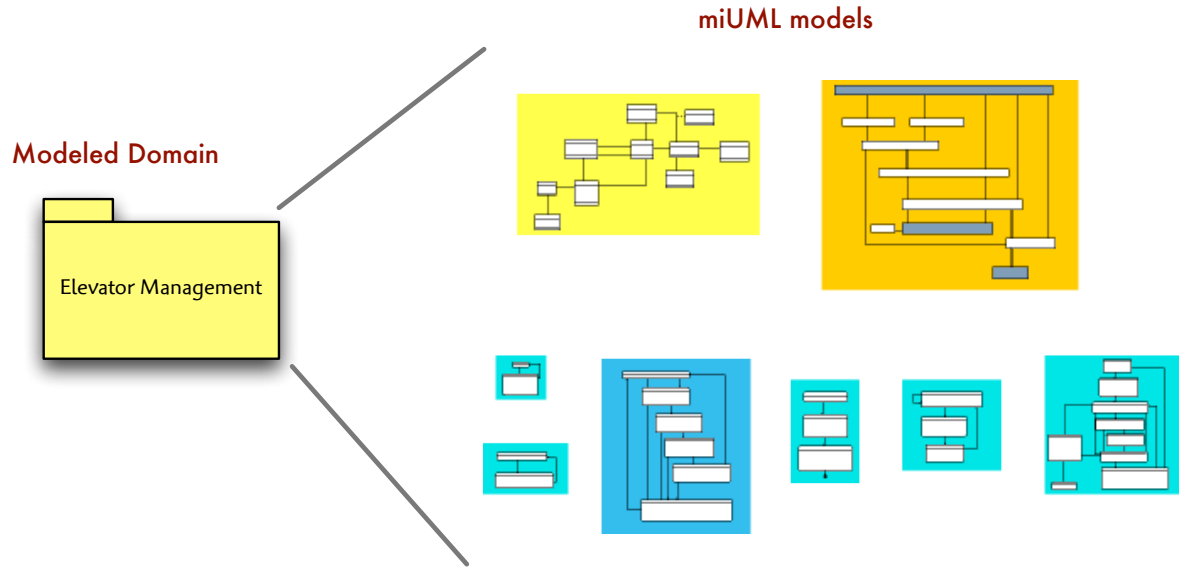
A Subsystem Element (Class, Relationship) is managed entirely within a single Subsystem whereas a Spanning Element (Domain Type, Lineage, Constrained Loop) cannot be confined to a single Subsystem.

While a Relationship may cross from one Subsystem to another, by policy, the Relationship is managed by either one or the other. While a Class may be referenced in many Subsystems, it is managed in only one designated Subsystem. In both cases consider the Element to originate as a component of a specific Subsystem.

This policy does not really make sense for Types which are available anywhere in a Domain. A Constrained Loop *could* conceivably be managed by Subsystem, but, as with Types, it is not clear that there would be any benefit to an arbitrary Subsystem designation.

Modeled Domain MDOM

There are many different modeling languages one can use to define all or part of a Domain. Since this miUML metamodel address the building blocks and rules for constructing a Domain using miUML, the term *modeled domain* refers specifically to a Domain modeled entirely using miUML.



A Modeled Domain is modeled with a complete set of miUML Class, State and Action models.

The important thing to understand about a Modeled Domain is that it constitutes a cohesive collection of Classes and Relationships. This represents an object-oriented partitioning of a system without regard to the implementation platform. A Domain may be implemented in its own task, or in fifty tasks, or possibly in one task with all of the other Domains in the system. A separate marking model may show how a given set of Modeled Domains are deployed onto physical execution units. This same flexibility is not usually applicable to Realized Domains.

The complete absence of deployment consideration on a domain chart (a collection of Domains and Bridges for a particular system) is platform independent. This means that the organization of a domain chart should survive ongoing changes in the underlying platform technology. This, in turn, means that productive model development may begin within Domains independent of changes in the platform such as operating system selection, choice of network protocols, memory management schemes, GUI technology and so forth. Naturally, any Domain centered around a particular platform technology (memory management, for example) may in fact be affected, but at least the changes will be restricted to that one subject matter.

Attributes

Name

Domain.Name

Identifiers

1> Name

Relationships

None.

Realized Domain RDOM

A Domain defined using any language other than miUML is a *realized domain*. A Realized Domain my represent legacy software, existing platform technology, a programming language, a GUI package or some Domain to be constructed with a different modeling language or to be speci

Attributes

Name

Domain.Name

Identifiers

1> Name

Relationships

None.

Spanning Element SPAN_ELE

Certain Elements do not originate in a specific Subsystem. Rather than arbitrarily assign these to a Subsystem for management, they are recognized as Subsystem independent.

Domain Type was the primary motivation for this abstraction. Constrained Loop followed suit. Neither of these Classes really belongs within any particular Subsystem and it would be awkward to force modelers to gerrymander Elements among Subsystems to make this work.

Attributes

Number

Element.Number

Domain

Element.Domain

Identifiers

1> Number + Domain

Same as the superclass.

Relationships

R17

Spanning Element IS A Type, Lineage or Constrained Loop

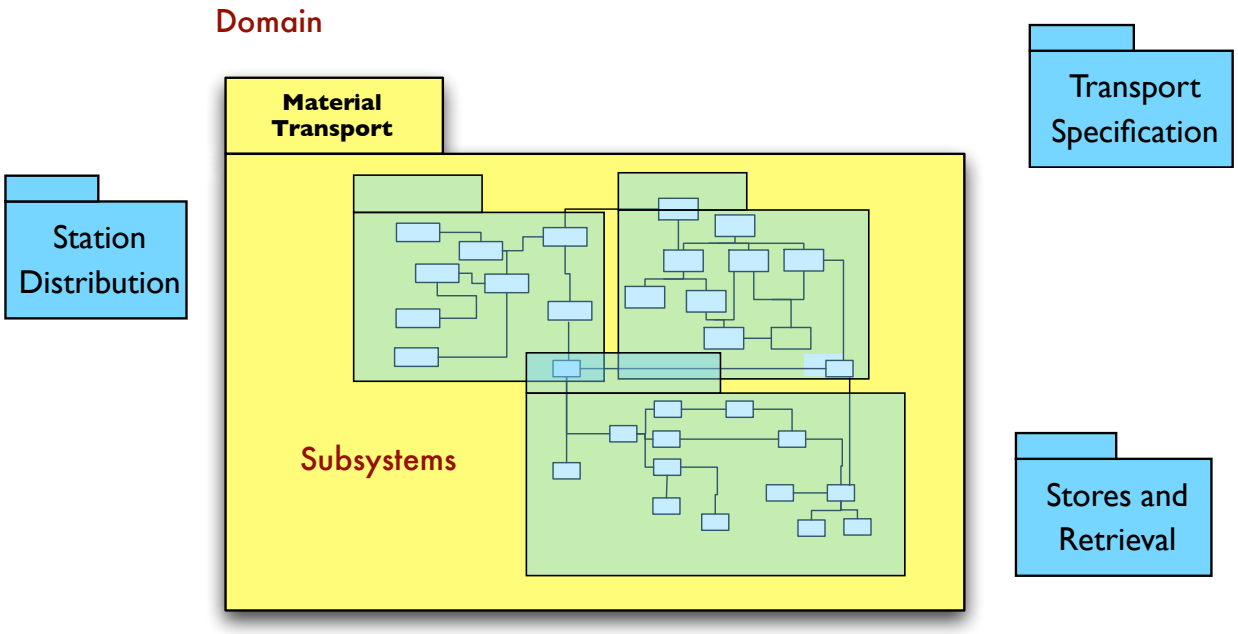
So far, these are the only three components of a Domain that flout Subsystem boundaries. There is no hard and fast rule to preclude the discovery of new subclasses. And it may be that Constrained Loop, someday, is subsumed as a special case of constraint which may or may not be Subsystem specific. But, for now, this seems to be the easiest way to manage these Elements.

A Type may be used by an Attribute in any Subsystem within a Domain. A Constrained Loop may include Relationship segments from multiple Subsystems. Finally, a Lineage may cross Subsystem boundaries since a Subsystem boundary may cross through one or more Generalizations.

Subsystem SUBSYS

"Many domains can be quite large with dozens or even hundreds of classes. Subsystems provide a way to partition a large domain so that people and groups can work with manageable chunks of the problem. A *subsystem* is a part of a domain containing classes, relationships, state machines, and their procedures." [MB]

miUML uses the definition in Executable UML above.



Subsystem boundaries cut across class model relationships.
As few as possible, ideally.

The same UML package symbol that represents Domains is used to represent Subsystems. And both Domains and Subsystems contain miUML model Elements. Beyond that, Domains and Subsystems are completely different.

A Domain is a coherent, complete subject matter. It is as big as it needs to be to cover the subject matter. A Subsystem is sized arbitrarily as a unit of management. One Subsystem grouping guideline could be 'whatever fits on a D-size sheet', for example.

Each Domain is a black box with respect to any other. So a Class, Attribute, State or any other modeled component in one Domain is invisible to external Domains. Within a Domain, Subsystems are white boxes. For example, an Attribute in one Subsystem may be read directly by an Element in another Subsystem (even though this might not be a good general practice). An Object in one Subsystem may receive and respond to Events from another Subsystem.

Ideally, these interactions are minimized by carving up Subsystems in such a way that as few Relationships and collaboration paths are divided as possible.

Attributes

Name

A descriptive name such as 'Stores and Retrieval', for example.

Type: Name

Alias

A short name used for labeling purposes. Known, in Shlaer-Mellor, as the keyletter, SAR, for example.

Type: Short Name

Domain

Domain.Name — The Subsystem manages content within this Domain.

Identifiers

1> Name + Domain

No two Subsystems within the same Domain may share names.

2> Alias + Domain

No two Subsystems within the same Domain may share aliases.

Relationships

R1
Subsystem MANAGES CONTENT OF *exactly one* Domain
Domain CONTENT IS MANAGED BY *one or many* Subsystem

In the minimal case, a Domain is managed as a single Subsystem. In this case, all Subsystem Elements are in the same Subsystem. Multiple Subsystems, though, may be defined with each Subsystem Element being assigned to one of these.

A Subsystem never extends outside of its Domain.

R3

Subsystem NUMBERS ELEMENTS WITHIN *exactly one* Subsystem Range
Subsystem Range ESTABLISHES NUMBERS AVAILABLE FOR *exactly one* Subsystem

A Subsystem Range is designated for each Subsystem. No Subsystem Range may overlap with another in the same Domain. Each subclass of Subsystem Element is numbered within the designated Subsystem Range.

Subsystem Element SUBELE

Elements such as Classes and Relationships can be readily assigned to a single Subsystem for definition and management. The Element is considered to be defined within a single Subsystem. Any changes to the Element or its descriptions are managed within the enclosing Subsystem.

Any Element managed by a specific, designated Subsystem is a Subsystem Element. For example, each Class must be assigned to a specific Subsystem.

Attributes

Number

Element.Number

Domain

Element.Domain

Identifiers

1> Number + Domain

Relationships

R14

Subsystem Element IS A Relationship or Class

So far, these are the only two components of a Domain managed by Subsystem. (Attributes are part of a Class and therefore indirectly managed by Subsystem).

Subsystem Range SRANGE

Each Subsystem uses a range of values its Elements. If a Subsystem uses the range 201-299, for example, each Class (Cnum) in that Subsystem will be numbered somewhere between 201 and 299 inclusive. The same is true for Relationships (Rnums). It is okay to have an Rnum and a Cnum with the same value in the same Subsystem.

Subsystem Ranges may not overlap. So if, for example, you have a Subsystem with the range 1-49, no other Subsystem may have a range beginning with a value less than 50.

In practice, it is much more common to refer to Rnums than Cnums. That is probably because Class names, having only one part, are easier to use than Association verb phrases. Some modelers ignore Cnums entirely.

Attributes

Floor

The lowest value that may be assigned within the range.

Type: Posint

Ceiling - constrained

The highest value that may be assigned within the range. Must be greater than or equal to the Floor value. (In practice, a range of only one number is rarely useful, but there is no reason to prevent it).

Type: Posint

Subsystem

Subsystem.Name — Numbers are assigned within this Subsystem.

Domain

Domain.Name — The Subsystem's Domain.

Identifiers

1> Floor + Subsystem + Domain

Ranges may not overlap within a Domain.

2> Ceiling + Subsystem + Domain

Same as above.

Relationships

None.