Platform Independent MDD Modeling Conventions

Version 3.0

January 21, 2009

PathMATE™ Series

Pathfinder Solutions
33 Commercial Street, Suite 2
Foxboro, MA 02035 USA

www.pathfindermdd.com
+1 508-568-0068

Table of Contents

1. Introduction

This document provides a summary of common rules and conventions applied to Platform-Independent Model Driven Development (PI-MDD) of complex systems. These rules and conventions have been developed to ensure the models are:

- Populated with clear, valid, and consistent abstractions
- Free of human errors caused by inconsistencies or unclear contexts
- Valuable to both the maintenance developer and the external expert reviewer

Please note that these conventions assume a base of syntactically correct models that pass all checks provided in the PathMATE Transformation Engine.

It is assumed that the reader is familiar with the PI-MDD approach as outlined in *Model Based Software Engineering: Rigorous Software Development with Domain Modeling*, Pathfinder Solutions. (This paper is available from www.pathfindermdd.com.).

For the best foundation in PI-MDD take the ***Effective MDD*** 5-day course from Pathfinder Solutions.

2. General Conventions

Naming and Descriptions

While the UML notation provides some semantic meaning through the graphic icons employed, there are many contexts where the true essence of specific abstractions is unclear. To help rapidly build an accurate understanding of the model for the model reader, the model authors must work to name model elements in a clear and creative way, and also capture an effective, appropriate and complete description.

It is helpful to narrow the scope of the naming/description problems: assume the reader is:

- Familiar with UML in general, with the general principles of PI-MDD, and are able either to navigate in your UML model environment or in the generated document they are using to read the model.
- Generally knowledgeable in the system concept of operations at a high level, and at least somewhat familiar with any specific domain's subject matter they may be reviewing.
- Not familiar with any of the specifics of the model you have constructed and not privy to (or does not remember) any specific modeling rationale you have applied.

Beyond their overview knowledge of the problem space you must either tell them (or reference documents that will tell them) everything else they will need to know to understand and perhaps even maintain/extend your model.

Effective model element names and descriptions are:

- Clear, concise, and unambiguous
- Appropriate to the subject matter of their domain

In addition, element descriptions should take care to avoid restating points apparent from the analysis itself. For example, avoid an attribute description that starts with "`This is the attribute of the Foo class which…`". Similarly names of model elements should not try to identify what type of model element it is, for example avoid names like `TargetClass` or `CTarget`, and `attr_BananaCount`.

Naming Conventions – Capitalization and Underscores

In contrast to disliked naming conventions `CTarget` and `attr_BananaCount` that consume both actual space in a name and also conceptual space in the mind, it is preferred to use a simple system of capitalization and underscores to help the model reader rapidly confirm a model element's type, especially at the Action Language level. (The rules are in `regular expression` syntax.)

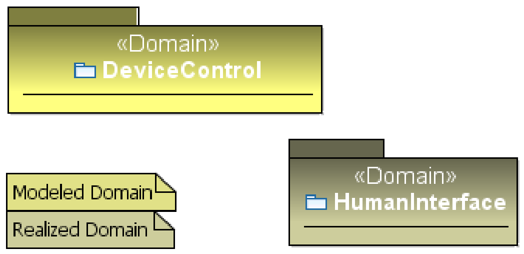
- DomainName, DomainServiceName, ClassName, StateName, SignalName: `[A-Z][A-Z,a-z,0-9]*`
- attributeName, classOperationName: `[a-z][A-Z,a-z,0-9]*` *(Some organizations choose to start static class operations names with a capital letter.)*
- data_type_name_t (for non-enum project-specific data types): `[a-z][a-z,0-9_]*_t`
- enum_name_e (for project-specific enumerates): `[a-z][a-z,0-9_]*_e`
- `SYMBOLIC_CONSTANT, ENUMERATE_LABEL: [A-Z][A-Z,0-9,_]*`
- action_local_variable, service_parameter, operation_parameter, signal_parameter: `[a-z][a-z,0-9_]*`
- association_role_phrase: `[a-z][a-z,0-9_]*`

Diagrams

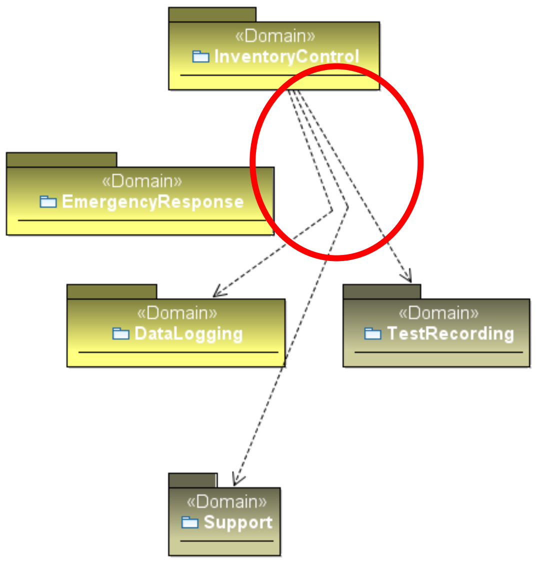
Software modeling affords the benefits of a two-dimensional form of expression – diagrams. A diagram can convey a lot of information in a small space – provided the needs of the reader are kept in mind.

- Diagram Descriptions – use these to identify organizational information your specific process may have you record, such as the include the author's name, and a document version number.

- The graphical layout groups related diagram elements together, and maintains uniform alignment and spacing to reinforce this grouping.
- Use colors to indicate major differences in model elements, and provide keys on the diagram:



- Care is taken in layout of connecting lines to avoid line crossover where possible.
- Connecting lines are routed with a minimum of intermediate nodes.
- When routing many (3 or more) connecting lines around a common obstacle, group or bundle them in a way to reduce clutter:



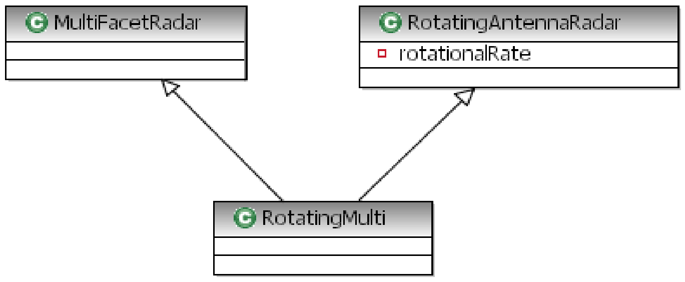
3. Diagram-Specific Conventions

Domain Model

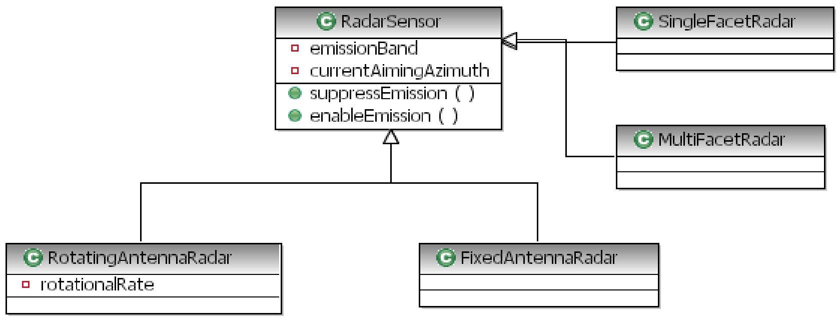
- Name domains to be reusable and independent of vendor. For example, choose *GUIToolkit*, not *MSVisualEnvironment*.
- Inter domain dependancy lines are straight, and at any angle.

Class Diagram

- Class names are singular, for example `Alarm`, not `Alarms`. Use a association multiplicity (`*`) to show plurality instead.
- Attributes are named from the perspective of their classes. This can afford some economy in the name, EG: for the `Hospital` class use `mainPhoneNumber` instead of `hospitalMainPhoneNumber`.
- Association names are `A<number>`.
- Avoid "multiple inheritance" generalization:

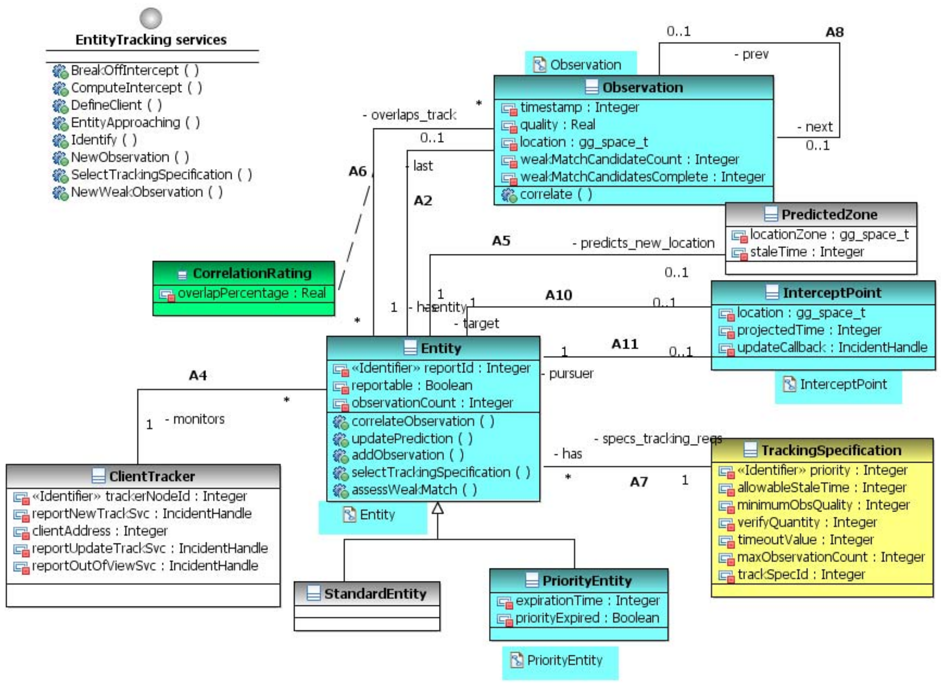


- Do not use disjoint generalization hierarchies. For example:



Instead use a discriminant attribute (where appropriate):

- Association and generalization lines are straight and orthogonal.
- Association names are located near the center of association lines.
- Association role phrases are located near their respective ends of their of association lines.
- Prefer verb-style role phrases to noun-style
- Always specify at least one role phrase
- Only specify a second role phrase (for the other participant) if it adds information to the model – avoid mechanically reflexive roles
- The supertype ends of generalization lines are aligned to present a single arrowhead.
- Use colors to identify major types of classes – white for plain, blue for active (have state machines), yellow for specification, and green for association:



Scenario Models

All

- Show a single logical path for each individual scenario model.
- Capture many scenarios to see what they may reveal, but only keep those that add meaningful new information to the core set you already have.
- Prefer to use asynchronous message arrows, unless the invoked domain service or class operation must return a value.

- Name each domain's interface class with "<*domain name*> services", or "<*domain prefix*> services" if space it at a premium.

System Level

- Only show lifelines for domains interface classes
- When drawing message lines, first create the domain service, quickly capture a preliminary definition for it, and then use it on the message line. Defer abstracting parameters unless they are essential to the meaning of the service.
- On message lines only show domain services. (Avoid free text or other.)

Domain Level

- Show the "local" domain interface as the first object lifeline on the left.
- Show server domains as the last object lifelines on the right.
- In general, show external impetus as coming from an external actor lifeline called "Client" at the far left to avoid specifying a specific client domain.
- Show only the following types of messages:
  - Domain service
  - Class operation
  - Class instance Create
  - Class instance Delete
  - Signal (generation)

State Model

- State names convey the lifecycle stage of a class instance for some finite span of time. Their names need to reinforce this, and present perfect tense can help. For example, use `RaisingRamp` instead of `RaiseRamp`.
- Signal names identify a single instance in time, and past tense can help. Their names can reinforce this. For example, use `DoorClosed` instead of `DoorClose`.
- Prefer Signal names that identify an incident or condition instead of a command: `TemperatureThresholdExceeded` instead of `StartCooling`.
- Use curved transition lines when available, or straight and orthogonal lines.
- Signal labels are located near the source end of their transition lines.

Actions

In specifying Actions it is important to follow some of the more general rules of structured programming:

- Only services returning a specific value need an explicit RETURN.
- When necessary, use a single RETURN statement.
- Reduce individual action scope to fit within 100 lines.
- Have a single statement on a line.
- Indent statement frames consistently within control structures.
- Provide liberal commenting, including:
  - ➢ summary at the top of the action, or in the state action summary
  - ➢ descriptions for control logic at each statement frame of a control structure (IF/ELSE, FOREACH, WHILE)
  - ➢ something for any line that is not patently self explanatory

Proper action modeling is necessary to enforce the rules and policies of the domain. Unconditional associations need to be enforced by the actions:

- When creating an instance of a class that has an unconditional relationship, the associated class instance should either be pre-existing or created shortly afterward, in the same action.
- Link unconditional associations as soon as possible after creating the participants, but always by the end of the action.

Even unconditional associations, however, do not guarantee successful navigation. An explicit LINK must have been completed before navigation:

- In all navigations, either provide defensive logic to check for NULL to handle “not found” cases, or explicitly comment where (action name) the link is guaranteed to have happened.

Separating the retrieval of an instance reference from its subsequent use can aid Design-level error catching, help with readability, and aid debugability:

- When invoking a service or method that returns an instance reference, catch this reference in a variable before using it. For example, instead of using this:

```
MO:PickBiggest().selected = TRUE;
```

do this:

```
Ref<MyObject> mo;
mo = MO:PickBiggest();
IF (mo != NULL)
{
```

```
mo.selected = TRUE;
}
```

User-Defined Types

In an effort to help the analyst understand where various types are defined and to afford a form of name-space scooping at the implementation level, the following naming conventions are applied to user defined types.

Scope Identification

- System-level data types are prefixed with “sys_”
- System-level symbolic constants and enumerate labels are prefixed with “SYS_”
- Domain-level data types are prefixed with “<domain prefix>_” (in lowercase)
- Domain-level symbolic constants and enumerate labels are prefixed with “<DOMAIN PREFIX>_” (in uppercase)

Enumerate Label Grouping with Enumerate Type

Even with defining scope prefixes on enumerate labels, it can be very difficult to identify which enumerate type definition and enumerate label is defined within without help. This convention helps in a type scope with many enumerate types:

- Enumerate label names are built from the scope prefix, and an identifying segment from the enumerate type name, followed by an identifying element from this value. Example: ENUM sys_traffic_light_color_e { SYS_TRAFFIC_LIGHT_RED, SYS_TRAFFIC_LIGHT_GREEN, SYS_TRAFFIC_LIGHT_BLUE}. *(Sorry – we’re out of yellow.)*