



# Platform Independent Action Language

Version 3.0  
February 2, 2009

---

*PathMATE*<sup>™</sup> Series

Pathfinder Solutions  
[www.pathfindermd.com](http://www.pathfindermd.com)  
+1 508-568-0068

# Table Of Contents

Preface.....	iii
PI-MDD and PathMATE Overview .....	iv
<b>1. Introduction.....</b>	<b>1</b>
<b>2. Action Overview .....</b>	<b>2</b>
What Are Actions? .....	2
What Can Actions Do?.....	2
What Makes Up an Action? .....	2
How are Actions Used?.....	3
<b>3. Action Semantics.....</b>	<b>4</b>
Data Context .....	4
Statements .....	5
<b>4. Readable Code and Actions .....</b>	<b>21</b>
<b>5. More Examples .....</b>	<b>21</b>
<b>6. One More Time - So How Come We Don't Just Put Code in the Model? .....</b>	<b>21</b>

# Preface

## *Audience*

The *Platform Independent Action Language Guide* is for modelers building Platform Independent models for transformation with the PathMATE environment. Familiarity with the Platform Independent Model Driven Development (PI-MDD) method and Modeling Language (UML) is helpful background knowledge for the material in this guide.

## *Related Documents*

These PathMATE documents are available at [www.pathfinder-mdd.com](http://www.pathfinder-mdd.com):

- *Accelerating Embedded Software Development with a Model Driven Architecture* (white paper)
- *PathMATE: Model Automation and Transformation Environment for Embedded Systems* (online brochure)
- *PathMATE Quick Start Guide*

## *Conventions*

The *PathMATE Modeler's Guide* uses these conventions:

- **Bold** is for clickable buttons, menu selections, and sub-headings.
- *Italics* is for screen text, path and file names, and other text that needs special emphasis.
- `Courier` denotes code, or text in a log or a batch file.
- A **Note** contains important information about a procedure or a process.

## *Getting Ready*

If you are not already familiar with the PathMATE toolset, read the overview that begins on page v. If you have not installed the PathMATE toolset on your computer, download the software from [www.pathfinder-mdd.com](http://www.pathfinder-mdd.com) and follow the installation instructions in available at the PathTECH portal at [www.pathfinder-mdd.com](http://www.pathfinder-mdd.com).

It is assumed that you are familiar with your UML editing tool. You may wish to complete the *PathMATE Quick Start Guide* for your UML tool.

# PI-MDD and PathMATE Overview

This overview introduces the Platform Independent Model Driven Development (PI-MDD) method and the PathMATE™ tools. PI-MDD is a coherent, end-to-end, step-by-step process for building special UML models for the deployment of high-performance software systems through fully automated and Self Optimizing code generation.

Through decades of industry experience, a this modeling methodology has evolved a carefully integrated set of mutually supporting architectural, modeling and deployment techniques. Building a complete and executable set of models completely at the problem-space level of abstraction, this approach has been refined and can now produce quantum gains in developer team productivity and delivered system quality. Based upon modern standards including the OMG Model Driven Architecture (MDA), automation technology makes the right way to build complex systems the fastest way as well.

## PathMATE Toolset

The PathMATE Model Automation and Transformation Environment includes all the tools required to transform your PI-MDD models into high-performance deployed systems (Figure 1).

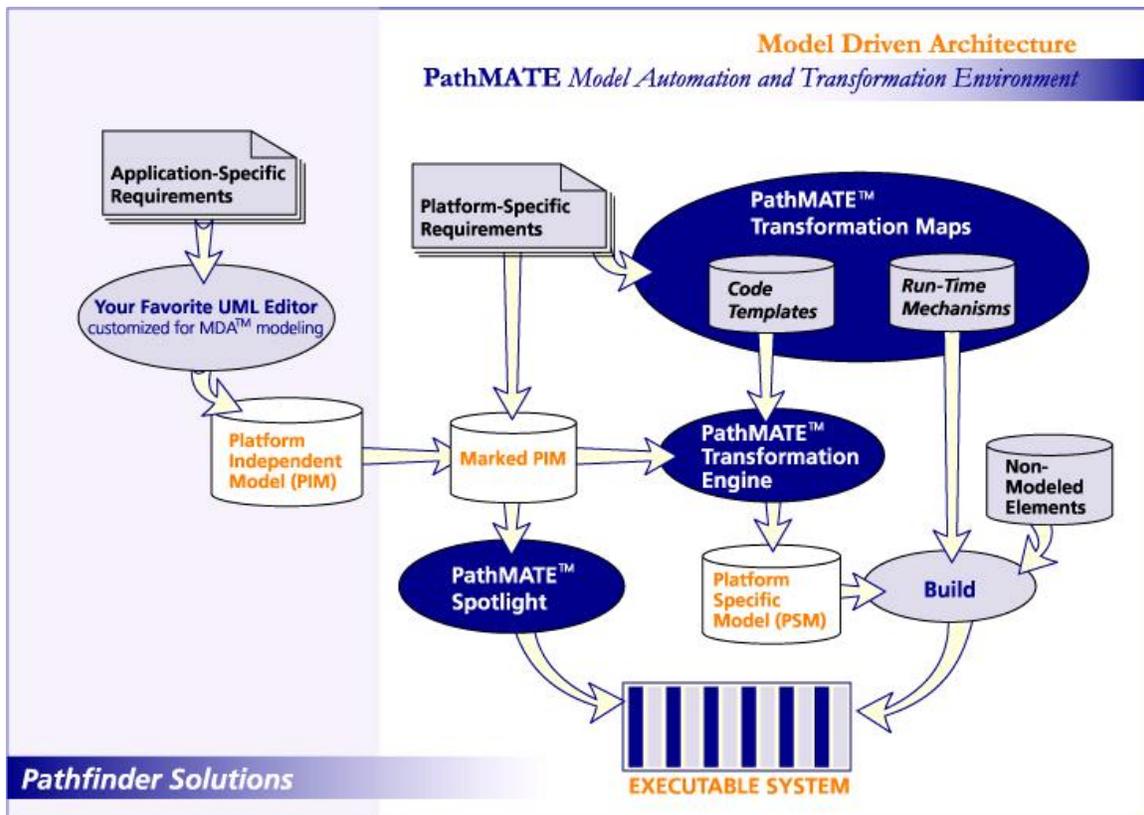
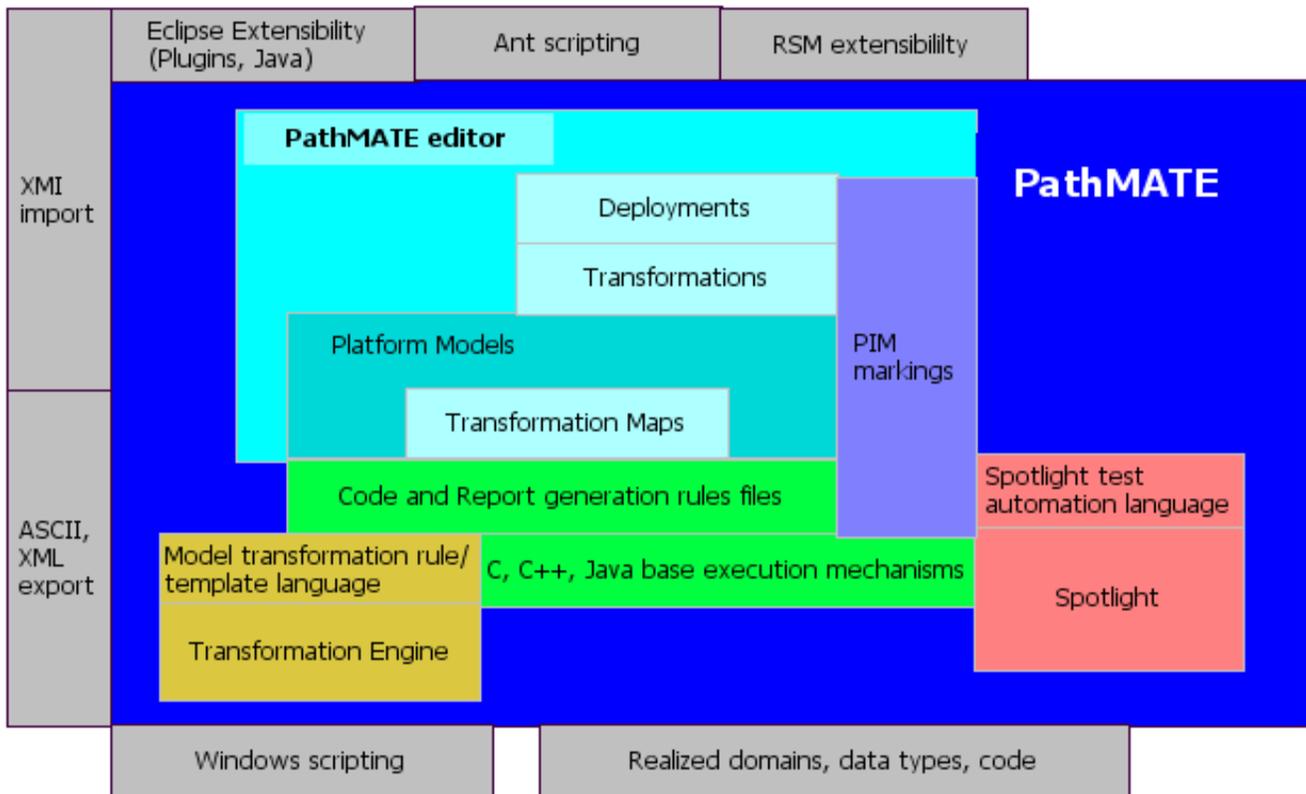


Figure 1. PI-MDD Methodology Flow

The three parts of the PathMATE toolset cooperate to turn your models into Self Optimized systems:

- *Transformation Engine* – The Engine extracts your model information from your UML tool, verifies consistency and correctness, and transforms it to a range of output forms via Transformation Maps.
- *Transformation Maps* – Generate C, C++, or Java implementation code with off-the-shelf Transformation Maps, or create custom maps to drive output for other languages or specific platforms.
- *Spotlight* – Verify and debug your application logic with Spotlight, the most advanced model-level execution and automated test environment available.

Based in the Eclipse environment, PathMATE is the most open and extensible model transformation technology available. This diagram shows all the aspects of PathMATE that allow process-level, transformation-level, code-level and other customizations to how this technology automates the development of your system.



---

# 1. Introduction

This document provides a summary of the Platform Independent Action Language (PAL) as applied in the PI-MDD method. PAL is a platform independent form of behavioral expression: a programming language for Platform Independent Models (PIMs) using the Object Management Group (OMG) UML Standard Action Semantics. Please see "UML Action Semantics Revised Final Submission," available from [www.omg.org](http://www.omg.org). Through its focus on platform independent model constructs, action language:

- Is concise and easy to learn, with a familiar, C++-like syntax.
- Provides the most convenient form of expression for PIM action procedures.
- Offers strategic agility through implementation platform independence and implementation language independence.
- Effectively enforces PIM separation from implementation code.
- Enables the highest degree of freedom to apply varying and project-specific implementation architecture and optimizations through transformation.

It is assumed that the reader is somewhat familiar with PI-MDD/UML modeling conventions as specified in the Model Based Software Engineering (MBSE) approach for modeling with the UML, as introduced in:

"Model Based Software Engineering: Rigorous Software Development with Domain Modeling," Pathfinder Solutions.  
(This paper is available from [www.pathfindermdd.com](http://www.pathfindermdd.com).)

The Action Language Quick Reference syntax summary is provided for your convenience in Appendix A. Print it separately and keep it handy.

---

## 2. Action Overview

### What Are Actions?

Actions are procedures in your UML models. More specifically, they are the operations, state actions and transition actions that specify the precise execution behavior of your modeled domains (PIMs):

- Domain service (interface class operation)
- Class operation
- State entry, exit and transition action
- System and domain initialization

### What Can Actions Do?

Actions operate within two realms. Actions from any context (from within any domain, or from the system initialization) can directly invoke the published services other domains. Actions (except for the system initialization) can also directly manipulate the model abstractions that exist within their own domain. This includes creation and deletion of class instances, reading and writing class attribute values, linking and unlinking of associations, generation of signals, and invocation of operations.

An action in one domain cannot access any internals within another domain.

### What Makes Up an Action?

An action is very similar in execution semantics to a function from a procedural programming language. It is made up of blocks of statements. The action itself has a root block of statements, and certain statements have blocks nested within them.

Each statement is made up of expressions and keywords. Expressions are accessors to Analysis elements, local variables, compound expressions (with operators), or literals.

---

## How are Actions Used?

Actions, along with all other Analysis elements in your system, are fed into the code generation step where they are mapped to executable implementation code. Since actions specify the complete behavior for a domain in model-level terms, PathMATE Transformation Maps producing implementation code carefully review how all model constructs are used by the actions to Self Optimize the code produced.

Unlike code-in-the-model approaches where actions are specified in implementation code, PAL in actions provide the information needed by the Transformation Maps to generate the most efficient code, automatically optimized for exactly how the model will behave:

- Self trimming of unused model elements
- Only build in run-time layer elements actually used
- Self-selection of optimal memory management, instance data storage and access mechanisms
- Generate tailored infrastructure code only when needed, avoiding unused or inefficient constructs
- Efficient topology resolution, automatically using local accesses where intertask and interprocessor mechanisms are not necessary.
- Transformation-time selection of optimal infrastructure, generating compile time resolution where possible

### 3. Action Semantics

Platform-independent Action Language (PAL) is a programming language with specific primitives to support the manipulation of Analysis elements. To describe Action Language syntax, this document uses the following conventions:

*[optional item]* { *either* | *or* } *0 or more iterations, ...*  
All bolded characters (such as **{ | }**, **[]** ) indicate actual use of these characters in the action language.  
*Italic* items are substitution items or annotations.  
All action language keywords are case sensitive, and are shown in **BOLD**.  
*//* In action language, comments are like in C++

```
// This is a commented action language statement  
Student.LastName = "Smith";
```

#### Data Context

Each action has a varying set of data atoms that it reads and/or writes.

##### *Explicit*

Each domain service or class operation may have parameters defined. Services and operations may also have a return value. State actions may have signal parameters. All actions may declare and use local variables.

##### *Implicit*

Instance-based class operations, or instance-based state actions have a "this" variable available as a Reference to the target instance. Literal and symbolic constant values may be used. FIND accessors over entire instance populations (FIND CLASS) imply the use of a domain-wide population of instance references.

##### *Data Types*

There is a fixed set of data atoms in a PIM: attribute, service, operation or signal parameter, and action local variable. Each atom is of a specific data type. There is a core set of basic, built-in data types:

- Boolean: TRUE or FALSE
- Character: an ASCII character
- Integer: whole number (width is design dependent)
- Handle: generic reference (similar to void\* in C)
- Real: floating point number (size is design dependent)
- FineGrainedTime: Specification of time to the nanosecond level
- String: a variable length ASCII string
- GenericValue: stores a String, Real, Handle, or Integer (similar to C union)

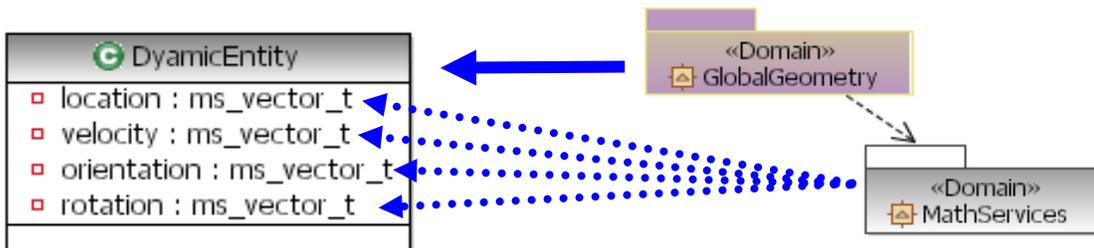
In addition there are advanced data types:

- Ref<class>: a reference to an instance of <class>, commonly used as a type for a local variable used to iterate over the results of a Find or Navigate
- Group<base type>: an ordered set of <base type> items, commonly used as a type for a service parameter to support passing sets of data items between domains
- GroupIter<base type>: an iterator over an ordered set of <base type> items, used to iterate over items in a group.

```
[ Boolean | Character | String | Real | Integer |
  GenericValue | Handle | Group<base_type> |
  GroupIter<base_type> | Ref<class_name> |
  FineGrainedTime | IncidentHandle]
```

The modeler can define new types:

- Enumerations
- Aliases base types (similar to typedef in C). The ms\_vector\_t below is an example of an alias of the base type Handle:



## Statements

Statements combine expressions to accomplish specific tasks within actions.

### Data Manipulation

**Assignment** - writes the value of the expression on the right of the equal side into the data atom on the left:

```
{ AttributeAccessor | Parameter | LocalVariable |
  IncidentHandleParameter } = Expression ;
```

```
Student.LastName = "Smith";
```

**Local Variable Declaration** – declares a local variable (scope limited to declaring action):

```
DataType variable_name { = initial_value };
```

```
String name; // declare name
```

```
Integer counter = 5; // declare counter - set to 5
```

**Constant Declaration** – in the system or domain initialization action declare a constant. Constants defined in the system initialization action are accessible to all domains. Constants defined in the domain initialization action are accessible only to the domain where they are defined.

```
CONST DataType variable_name = initial_value;
```

```
CONST INTEGER EMIF_MAX_RECEIVE_BUFFER = 1024;
```

**External Constant Declaration** – in the system or domain initialization action declare a constant that is defined in realized code. The action language parser will recognize an external constant but will not create a definition for it. External constants defined in the system initialization action are accessible to all domains. External constants defined in the domain initialization action are accessible only to the domain where they are defined.

```
EXTERN CONST DataType variable_name;
```

```
EXTERN CONST INTEGER EMIF_MAX_RECEIVE_BUFFER;
```

**Note:** Some Transformation Maps such as the Pathfinder C++ Map support an IncludeFile property that contains the name of a realized include file containing the definition of the external type. Consult the Design User's Guide for more information.

**Data Atom Ordering** - sorts the specified list of data atoms based on their value. "/" indicates ascending order (default), or "\" indicates descending order:

```
ORDER GROUP [{ / | \ }] group;
```

**Instance List Ordering** – These statements sort the specified class instance population based on the specified attribute(s). The attributes can be preceded by "/" to indicate ascending order (default), or "\" to indicate descending order. The most significant key is specified first:

**Class Population Ordering** – sorts the specified class instance population:

```
ORDER CLASS class_name BY ([{ / | \ }]
attribute_name, ...);
```

Sensor	
□	dataSourceId : Integer
□	detectionZone : Integer
□	position : gg_coord_t
□	operationalState : Integer
●	initialize ( )

```
// order all instances of Sensor ascending by
// detection zone
ORDER CLASS Sensor BY (/detectionZone);
```

**Note:** Data ordering is not maintained when new elements are added to the group or instance population. For example, if an ORDER statement was executed and a subsequent action created a new instance of the class, the sort order specified by the ORDER statement would not be maintained. If you want a sort order to be maintained, use the Sort design properties.

**Execution Flow Control**

Action Language contains conditional and iterative execution flow control constructs.

**Statement Block** – A statement block is a sequence of statements.

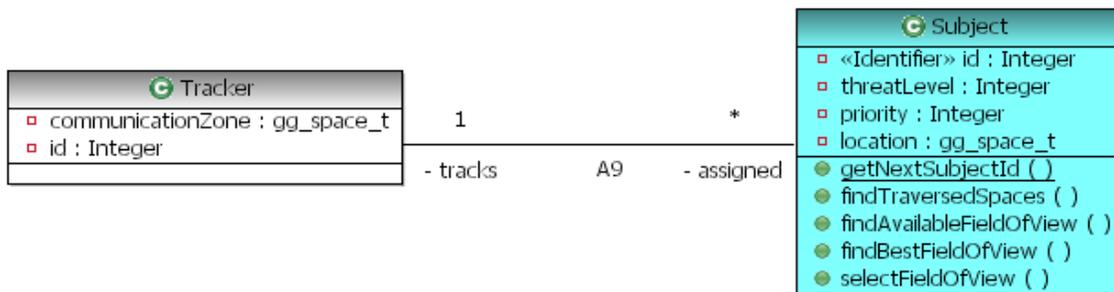
**Instance List Iteration** – These statements declare a cursor variable, and then iterate over each class instance in the specified population. Each instance is assigned to the cursor variable, and the nested statement block is executed. An optional WHERE clause filters the instance set to only those that match the specified Boolean expression comparing attributes of the target class with any data atoms available in the action context.

**Class Population Iteration** - iterate over the entire class population:

```
FOREACH cursor_variable = CLASS class name [ WHERE
(Expression) ] { StatementBlock }
```

**Association Population Iteration** - iterate over the associated instance population (Navigation is an association navigation expression):

```
FOREACH cursor_variable = Navigation [ WHERE
(Expression) ] { StatementBlock }
```



```
// for each threat (any level) assigned to
gf_tracker
FOREACH threat = gf_tracker -> A9
WHERE (Subject.threatLevel > SRM_MIN_THREAT)
{
```

```
body of loop...
}
```

**Conditional** – execute the first statement block if the Boolean Expression is TRUE, otherwise execute the optional else statement block.

```
IF (Boolean Expression) { StatementBlock }
[ ELSE IF (Boolean Expression) {StatementBlock} ]
[ ELSE { StatementBlock } ]
```

**Iterative** – evaluate the specified Boolean expression – if it is TRUE, execute the statement block. Repeat until the expression is FALSE, or a BREAK statement is encountered.

```
WHILE (Expression) { StatementBlock }
```

**Break** – interrupt execution of the enclosing iterative control structure (WHILE or FOREACH). Skip all remaining statements in the iterative statement block, and resume execution after the iterative statement block:

```
BREAK;
```

**Continue** - interrupt execution of only this iteration of the enclosing iterative control structure (WHILE or FOREACH). Skip all remaining statements in the iterative statement block, and resume execution at the top of the iterative statement block:

```
CONTINUE;
```

### Function Ins and Outs

**Invocation** – call the specified service, operation or built-in method (with no return value):

```
{ Service Accessor | BuiltIn method};
```

```
// Domain service
EntityTracking:Identify(client_id, local_id);
```

**Service/Operation Value Return** – return from this service with the specified return value:

```
RETURN [ Expression ];
```

### Expressions

An expression is something that provides a data value, receives a data value (when it is written to), and/or performs some action. Expressions are used to create, store and access data values and Analysis elements. Expressions are also used to invoke services and access built-in capabilities.

## Variables

Local Variable Reference – used after its declaration:

*variable\_name*

Parameter – service parameters can be referenced in the service action; signal parameters can be referenced in the state action:

*parameter\_name*

## Constants

Constant Reference – system or domain scoped constants defined in the initialization action may be used within the scope of the constant:

*constant\_name*

## Accessors

Accessors are expressions that read or write specific Analysis data atoms.

### Class and Attribute Accessors

**Attribute** – read or write a class attribute value. The *instance\_ref* can be a local variable, a service or signal parameter, or the “this” reference in instance-based class operations or states:

```
instance_ref . attribute_name
```

```
// Read attribute from some other instance
new_subj_type = new_subject.type;
// Write attribute in this instance
this.type = SRM_CONFIRMED_TRACK;
// Same as above
type = SRM_CONFIRMED_TRACK; // "this" assumed
```

**Class Instance Create** – create an instance of the specified class and return a reference to it. A leaf subtype must be specified (no supertypes). Attribute values must be specified if there is no default. An initial state name must be specified if there is no default (a default state is specified by the initial state bullet on a state model):

```
CREATE class_name ( [ attribute_name = Expression,
... ] ) [ IN initial_state ]
```

```
// create a field of view instance
Ref<FieldOfView> fov = CREATE FieldOfView (id =
provided_id, space = target_space);
```

**Class Instance Delete** – unlink the specified instance from all associations it participates in, and remove it:

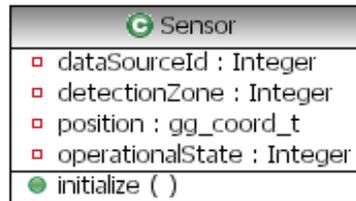
```
DELETE instance_ref
```

```
DELETE fov;
```

**Note:** PathMATE Transformation Maps automatically unlink a deleted instance from all its associations.

**Class-Based Find** – Find the first (default) or last instance of the specified class. An optional WHERE clause filters the instance set to only those that match the specified Boolean expression comparing attributes of the target class with any data atoms available in the action context. If no matching instance is found, NULL is returned:

```
FIND [ { FIRST | LAST } ] CLASS class_name [ WHERE
(Expression) ]
```



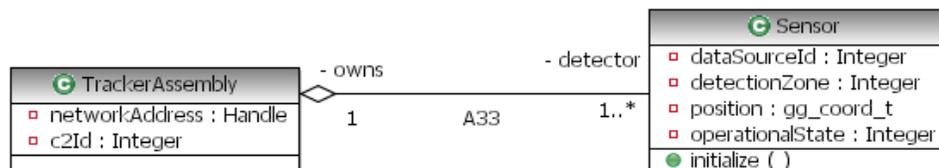
```
// search all sensor instances and
// find number 314
Ref<Sensor> detector_314 = FIND CLASS Sensor
WHERE(Sensor.dataSourceId == 314);

// find any sensor
Ref<sensor> any_sensor;
any_sensor = FIND CLASS Sensor;
```

**Navigation-Based Find** – Find the first (default) or last instance through the specified chain of association navigations. An optional WHERE clause filters the instance set to only those that match the specified Boolean expression comparing attributes of the target class with any data atoms available in the action context. If no matching instance is found, NULL is returned:

```
FIND [ { FIRST | LAST } ] Navigation [ WHERE
(Expression) ]
```

**Note:** WHERE expressions are Boolean expressions that can include any data atom available from the action context: local variables, constants, and parameters. It also includes attributes of the target instance. However a WHERE expression cannot perform other instance-based accesses of the target instance, including instance-based operation invocations, or association accesses (navigation).



```
// locate a Sensor in fault state
```

```
// (We are in a TrackerAssembly action)
Ref<Sensor> faulted_detector;
faulted_detector = FIND FIRST this -> A33
WHERE(Sensor.operationalState == SI_FAULT);
```

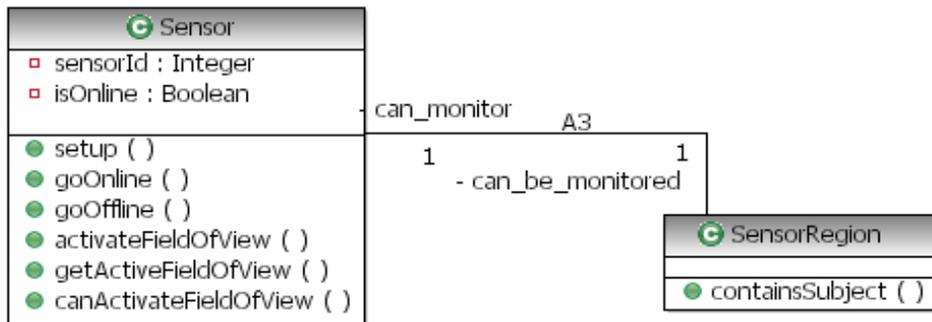
### Relationship Accessors

**Association Link** – establish a connection between the specified class instances. If the association is reflexive (the same class at both ends), then at least one role phrase must be specified. If the association has an associated class, an associated class instance reference must be provided.

```
LINK [ @role_phrase1 ] instance1_ref
A<number> [ @role_phrase2 ] instance2_ref
[ASSOCIATIVE assoc_ref ]
```

**Unlink** - break the connection between the specified class instances. If the association is reflexive (the same class at both ends), then at least one role phrase must be specified. If the association has an associated class instance connected, this instance is deleted automatically by unlink. Do not delete the associated class instance prior to or after the unlink.

```
UNLINK [ @role_phrase1 ] instance1_ref
Anumber [ @role_phrase2 ] instance2_ref
```



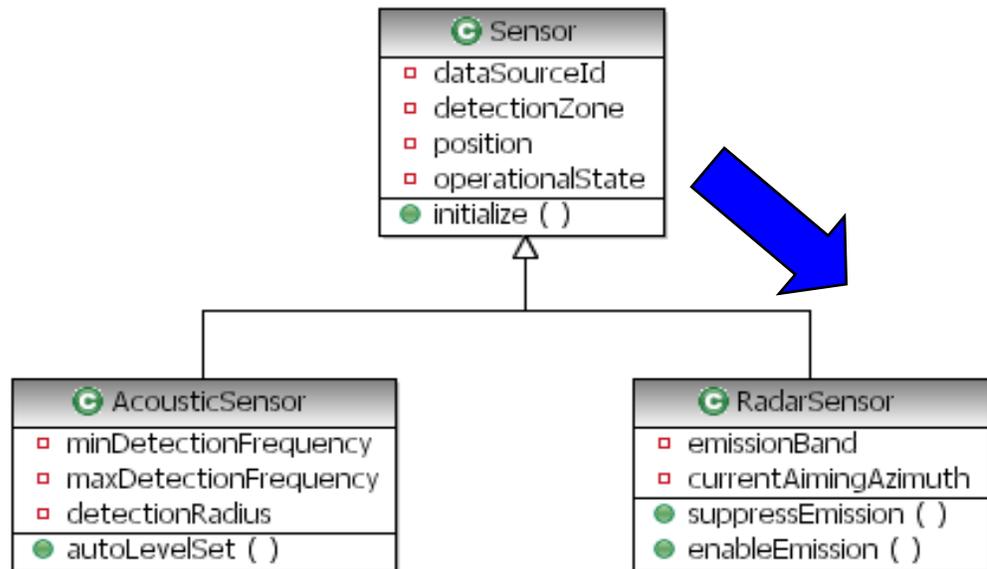
```
Ref<SensorRegion> old_region;
Ref<SensorRegion> new_region;
Ref<Sensor> sensor;
.
.
UNLINK sensor A3 old_region;
LINK sensor A3 new_region;
```

**Navigation Expressions** – There are used to traverse associations and inheritance relationships (downward only). A Navigation Expression may return no instances, a single instance, or a collection of instances – all depending on the multiplicity of the associations in the Navigation Expression. A Navigation Expression may contain multiple individual navigations chained together with the across operator “->”. A Navigation Expression cannot be used as a class

instance reference expression itself – it is only used in the context of a FIND accessor, FOREACH statement, or ORDER statement.

**SubSuper Navigation** – “downcast” to get from a supertype to a specific subtype. Returns NULL if the actual subtype encountered at run-time subtype does not match specified subtype. Upcasting is performed automatically. A subtype can be used anywhere a supertype is expected.

```
supertype_reference -> subclass_name
```



```
// Ref<Sensor> asset
// navigate from asset to
// RadarSensor subtype
asset -> RadarSensor
```

**Association Binary Navigation** – Navigate from the start\_ref class instance across the specified association to the instance(s) at the other end:

```
[ @role_phrase1 ] start_ref->A<number> [ -
>@role_phrase2 dest_class_name]
```

**Association Navigation to Associated Class** – Navigate to the instance of the class associated with a link between class instances start\_ref\_1 and start\_ref\_2:

```
[ @role_phrase1 ] start_ref_1 AND [ @role_phrase2 ]
start_ref_2 ->A<number>
```

### Signal

**Generate** – create an instance of the specified signal, and queue it for dispatch to the specified instance. No destination is provided for create

signals. Destination is optional for self-directed signals sent to self from an instance state. All signal parameters must have a value provided. If a delay is specified, the signal will be held in the delayed signal mechanism for a minimum of the period specified, and then it will be queued for dispatch. Delay can be specified in via a `FineGrainedTime` with its nanosecond precision, or if an integer is used the units are assumed to be milliseconds:

```
GENERATE signal_name ( Expression, ... ) [ AFTER
(delay) ] [ TO (destination_ref) ]
```

```
// self-directed signal - defaults TO (this)
GENERATE Subject:PositionUpdated(new_pos);

// signal to another instance
GENERATE Subject:PositionUpdated(new_pos) TO
(priority_subject);

// delayed signal TO (this)
GENERATE Subject:RetryTimerExpired() AFTER
(SRM_RETRY_MS) TO (subj);
```

**Cancel** – If an instance of this signal destined to the specified destination is still held in the delayed signal mechanism, then remove it before transmission. If more than one instance of this signal is outstanding against the specified destination, delete the one with the shortest delay remaining. No indication is returned if this operation actually found an instance of the signal.

```
CANCEL signal_name [ TO (destination_ref) ]
```

```
// Set timeout
GENERATE Subject:RetryTimerExpired() AFTER
(SRM_RETRY_MS) TO (subj);
. . .
// cancel timeout
CANCEL Subject:RetryTimerExpired TO (subj);
```

## Service and Operation Invocation

**Domain Service Invocation** – may have a return value:

```
domain_prefix:service_name(Expression, ...)
```

```
// Domain service
EntityTracking:Identify(client_id, local_id);
```

**Class operation Invocation (class based)** – may have a return value:

```
class_prefix:service_name(Expression, ...)
```

**Class operation Invocation (instance based)** – may have a return value:

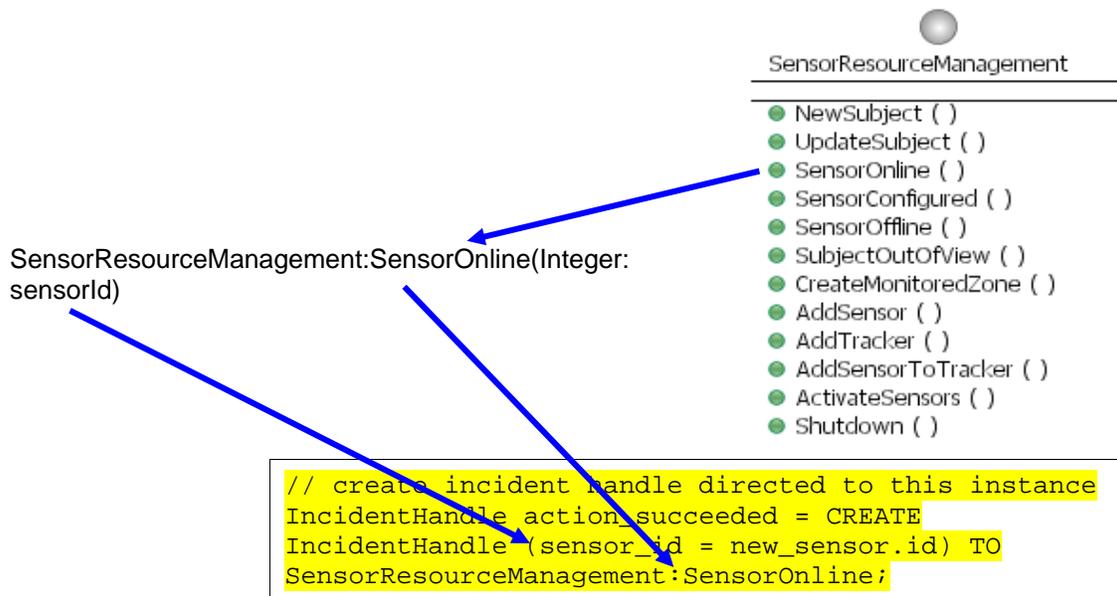
```
instance_ref . [class_prefix:
]service_name(Expression, ...)
```

**Note:** Always omit the class prefix when calling a polymorphic service on a supertype instance.

### IncidentHandle

*Create IncidentHandle* – create a IncidentHandle to a specific service (only valid to services in context domain). Input parameter values must be specified if there is no default:

```
CREATE IncidentHandle ( [ parameter_name =
Expression, ... ] ) TO { domain_prefix | class_prefix
}:service_name
```



**Invoke IncidentHandle** – input parameter values are optionally specified:

```
CALL service_handle ( [ parameter_name =
Expression, ... ] )
```

**IncidentHandle Parameter** – a IncidentHandle parameter can be directly referenced for read or write using the parameter name as an index (error behavior is Design-specific):

```
service_handle [ parameter_name ]
```

```
client_callback [ status ] = NO_ERROR;
```

---

**NOTE:** Using parameter values requires the specification of an IncidentHandle Profile to

- Name to parameter set
- Ensure correspondence between set and expected parameters values
- Permit the server to use an Incident Handle independent of the client

The client-side service is marked with a Profile Name. The data elements on the service side holding the Incident Handle are marked with the same name. The specification of Incident Handle Profiles is a UML editor-dependent activity, and is documented in [/pathmate/doc/TechNotes/TechNote\\_IncidentHandleProfile.pdf](/pathmate/doc/TechNotes/TechNote_IncidentHandleProfile.pdf)

### **Built-In methods**

**Invocation** – built-in methods are invoked as methods of their operand:

```
Operand.method_name(parameters...)
```

**Group built-ins** - Group data types have the following support methods:

Add an item in front of the first item in the group:

```
group_expression.addFront(item)
```

Add an item after the last item in the group:

```
group_expression.addBack(item)
```

Add an item after the current position (indicated by the iterator):

```
group_expression.insert(iter, item)
```

Return the first item in the group:

```
group_expression.front()
```

Return the first item in the group:

```
group_expression.back()
```

Remove the first item with the specified value from the group:

```
group_expression.remove(item)
```

Delete the item at the iterator location from the group:

```
group_expression.erase(iter)
```

Remove all items from the group:

```
group_expression.removeAll()
```

Return an integer specifying the number of items in the group:

```
group_expression.size()
```

Return the specified item in the group (0-based index; error behavior is Design-specific):

```
group[index]
```

```

Group<String>  names;

// adding elements
names.addBack("Joe"); // Joe
names.addFront("Mary"); // Mary, Joe
names.addBack("Sue"); // Mary, Joe, Sue

// accessing elements
String name1 = names.front(); // Mary
String name2 = names[1]; // Joe
String name3 = names.back(); // Sue

```

**GroupIter built-ins** - GroupIter data types have the following support methods:

Establish the base group for the iterator – required before any other iterator operations:

```
group_iter_expression.setGroup(group)
```

Reset the iterator to the front of the list:

```
group_iter_expression.front()
```

Return the iterator to the back of the list:

```
group_iter_expression.back()
```

Return the current item in the group:

```
group_iter_expression.current()
```

Increment the iterator's position in the group, and return the new current item in the group:

```
group_iter_expression.next()
```

Decrement the iterator's position in the group, and return the new current item in the group:

```
group_iter_expression.previous()
```

Return a boolean indicating if the last next() operation has advanced past the end of the list, or if the last previous() operation has advanced past the beginning of the list:

```
group_iter_expression.finished()
```

```

Group<String>  names;

// iterate from beginning to end
GroupIter<String>  cursor;
cursor.setGroup(names);
cursor.front();

```

```
WHILE(!cursor.finished())  
{  
    // get name at current iterator position  
    String current_name = cursor.current();  
    // ... do something with name  
    // advance to the next item in the group  
    cursor.next();  
}
```

## Expression Mechanics

**Binary Expression** – has two operand expressions combined by an operator. Binary expressions can be nested, and grouped with parenthesis:

```
[ ( ] Expression Operator Expression [ ) ]
```

### Arithmetic Binary Operators

+ plus  
- minus  
\* multiply  
/ divide  
% modulus

### Bitwise Binary Operators

& and  
^ exclusive or  
| inclusive or  
<< left shift  
>> right shift

### Boolean Binary Operators

< less than  
<= less than or equal to  
> greater than  
>= greater than or equal to  
&& and  
|| or  
== equal to  
!= not equal to

### Unary Expression –

```
[ ( ] UnaryOperator Expression [ ) ]
```

### Unary Operators

+ arithmetic positive  
- arithmetic negative  
~ complement  
! Boolean not

## Literals

### Boolean Literal -

```
{ TRUE | FALSE }
```

### Character Literal – a single character:

```
'character'
```

### Integer Literal - one or more digits:

```
digit...
```

```
123 or 0xffffffff
```

### Invalid Class Instance Reference -

```
NULL
```

### Uninitialized or Invalid IncidentHandle Reference -

```
EMPTY_SERVICE_HANDLE
```

### Real Literal

```
IntegerLiteral . IntegerLiteral [ e [ - ]  
IntegerLiteral -
```

```
3.45 or 3.45e-6
```

### String Literal - use \ to embed a double quote " :

```
"character..."
```

## Attaching Marking Values to Statements

**Marking Values** – each PAL statement may have one or more PIM markings applied in name value pairs accessible to the Transformation Maps. Properties are defined in curly braces after the statement semicolon or closing curly brace. The property name must begin with a letter followed by a letter, number or underscore.

```
statement ; [{ property_name = "property_value",  
...}]
```

```
WD:DetectWeatherElements (input_buffer); {Routing= SIGNAL_CONDITIONING}
```

## 4. Readable Code and Actions

The *Platform Independent MDD Modeling Conventions* available in [/pathmate/doc/modeling\\_conventions.pdf](/pathmate/doc/modeling_conventions.pdf) provide a wealth of guidelines on techniques that experienced teams use to build modeler-friendly models. Central is the set of naming conventions that can be especially helpful to the writers/readers of PAL actions. We encourage you to adopt them (or some variant of your own).

## 5. More Examples

The best place to see PAL examples is in each of the fully executable sample systems provided with PathMATE, including SimpleOven, ExperimentControl and others (depending on your UML editor integration with PathMATE).

## 6. One More Time - So How Come We Don't Just Put Code in the Model?

Some of the most popular MDD tools support code-centric development. This is a great way for organization sensitive to cultural change ease their way into the world of modeling. But code-centric models have been shown to be twice as complex as PI-MDD models. Specifically in the case of actions, implementation languages have critical limitations:

- They do not directly support manipulation of UML primitives
- It takes work to define and implement a portability layer
- This type of action programming builds dependencies on a specific configuration of deployment topology
- Implementation strategies/patterns are hard coded and cannot be readily changed after coding starts
- It's hard to learn how to write fast and safe C++ code. It is harder to successfully teach others how. Perhaps hardest of all is ensuring they always do it right, each time, each update.
- Implementation code is at the wrong level of abstraction for models.

Platform-independent Action Language directly addresses these issues:

- Simple, extensible
- Direct support for UML at the PIM level
- Eliminates unsafe and inefficient constructs
- Platform independent:
- Rapid porting to new target environments
- Deployable to any topology – without change.

From the perspective of the development of high performance systems, perhaps the most critical features PAL provides:

- It is translated to code, so the implementation strategy/patterns can be updated and optimized at any time

- Open transformation technology facilitates project-unique optimizations.

But even beyond the technical benefits, PI-MDD is a response to business drivers:

- It is easier to learn PAL than common implementation languages
- PAL is commercially available – you don't have to invest in home-grown definition/invention/review processes and automation
- PAL conforms to industry (OMG) standards

In the end it is the Bottom Line that trumps the rest:

- PAL is the key to PI-MDD, which drives productivity 2x and higher.

## PAL SYNTAX SUMMARY

[optional item] {either | or} 0 or more iterations, ...

All bolded characters (such as {}, []) indicate actual use of these characters in PAL. *Italic* items are substitution items or comments.

All PAL keywords are case sensitive; // comments like C++; An Action Procedure has signal or service Parameters and a StatementBlock.

### Statement

Assignment	{ AttributeAccessor   Parameter   LocalVariable   IncidentHandleParameter } = Expression ;
Break	<b>BREAK;</b>
Continue	<b>CONTINUE;</b>
LocalVarDecl	DataType <i>variable_name</i> { = <i>initial_value</i> };
ConstantDecl	[ <b>EXTERN</b> ] <b>CONST</b> DataType <i>variable_name</i> [ = <i>initial_value</i> ]; - valid only in system or domain init. action
Invocation	{ Accessor   Builtin method }; - a procedure call with no return value
ForEach	(class) <b>FOREACH</b> <i>cursor_variable</i> = <b>CLASS</b> <i>class_name</i> [ <b>WHERE</b> (Expression) ] { StatementBlock }
ForEach	(nav) <b>FOREACH</b> <i>cursor_variable</i> = Navigation [ <b>WHERE</b> (Expression) ] { StatementBlock }
If	<b>IF</b> (Boolean Expression) { StatementBlock } [ <b>ELSE IF</b> { StatementBlock } ] [ <b>ELSE</b> { StatementBlock } ]
Return	<b>RETURN</b> [ Expression ];
StatementBlock	Statement...
While	<b>WHILE</b> (Expression) { StatementBlock }
Order	(class) <b>ORDER CLASS</b> <i>class_name</i> <b>BY</b> [{ /   \ }] <i>attribute_name</i> , ...;
Order	(nav) <b>ORDER</b> Navigation <b>BY</b> [{ /   \ }] <i>attribute_name</i> , ...;
Order	(group) <b>ORDER GROUP</b> [{ /   \ }] <i>group</i> ;

### Accessors

AttributeAccessor	<i>instance_ref</i> . <i>attribute_name</i>
CreateIncidentHandle	<b>CREATE IncidentHandle</b> ( [ <i>parameter_name</i> = Expression, ... ] ) <b>TO</b> { <i>domain_prefix</i>   <i>class_prefix</i> } : <i>service_name</i> - returns a service handle
GroupItemIndex	<i>group</i> [ <i>index</i> ] - 0-based index
InvokeIncidentHandle	<b>CALL</b> <i>service_handle</i> ( [ <i>parameter_name</i> = Expression, ... ] ) - no return
DomainServiceInvocation	<i>domain_prefix</i> : <i>service_name</i> (Expression, ...)
ClassServiceInvocation	<i>class_prefix</i> : <i>service_name</i> (Expression, ...)
InstanceServiceInvocation	<i>instance_ref</i> . [ <i>class_prefix</i> : ] <i>service_name</i> (Expression, ...)
SubSuperAccessor	<i>supertype_reference</i> -> <i>S</i> <i>relationship_number</i> -> <i>subclass_name</i> - may have a return value
AssociationAccessor	[ Link   Navigate   Unlink ] - performs a "downcast"
Link	<b>LINK</b> [ @ <i>role_phrase1</i> ] <i>instance1_ref</i> <b>A</b> <number> [ @ <i>role_phrase2</i> ] <i>instance2_ref</i> [ <b>ASSOCIATIVE</b> <i>assoc_ref</i> ] - no return
Navigation	(binary) [ @ <i>role_phrase1</i> ] <i>start_ref</i> -> <b>A</b> <number> [ ->@ <i>role_phrase2</i> <i>dest_class_name</i> ]
Navigation	(to assoc class) [ @ <i>role_phrase1</i> ] <i>start_ref_1</i> <b>AND</b> [ @ <i>role_phrase2</i> ] <i>start_ref_2</i> -> <b>A</b> <number>
Navigation	(downcast) <i>supertype_instance_ref</i> ->< <i>subclass_name</i> >
Unlink	<b>UNLINK</b> [ @ <i>role_phrase1</i> ] <i>instance1_ref</i> <b>A</b> number [ @ <i>role_phrase2</i> ] <i>instance2_ref</i>
SignalAccessor	{ Cancel   Generate   ReadTime }
Cancel	<b>CANCEL</b> <i>signal_name</i> [ <b>TO</b> ( <i>destination_ref</i> ) ]
Generate	<b>GENERATE</b> <i>signal_name</i> ( Expression, ... ) [ <b>AFTER</b> ( <i>delay</i> ) ] [ <b>TO</b> ( <i>destination_ref</i> ) ]
ClassAccessor	{ Create   Delete   Find }
Create	<b>CREATE</b> <i>class_name</i> ( [ <i>attribute_name</i> = Expression, ... ] ) [ <b>IN</b> <i>initial_state</i> ] - returns inst. reference
Delete	<b>DELETE</b> <i>instance_ref</i>
Find	(class) <b>FIND</b> [ { <b>FIRST</b>   <b>LAST</b> } ] <b>CLASS</b> <i>class_name</i> [ <b>WHERE</b> (Expression) ] - returns an inst. reference or NULL
Find	(nav) <b>FIND</b> [ { <b>FIRST</b>   <b>LAST</b> } ] Navigation [ <b>WHERE</b> (Expression) ] - returns an inst. reference or NULL

### Expression

<i>accessors</i>	AttributeReference, SignalAccessor, ClassAccessor, AssociationAccessor, SubSuperAccessor
BooleanLiteral	BinaryOpExpression, Expression Operator Expression
CharacterLiteral	{ <b>TRUE</b>   <b>FALSE</b> }
IntegerLiteral	' <i>character</i> '
<i>invalid reference</i>	<i>digit</i> ... - one or more digits
LocalVariable	<b>NULL, EMPTY_SERVICE_HANDLE</b>
Parameter	<i>variable_name</i>
RealLiteral	<i>parameter_name</i>
IncidentHandleParameter	IntegerLiteral . IntegerLiteral [ <b>e</b> [ - ] IntegerLiteral ] - 3.45 or 3.45e-6
StringLiteral	<i>service_handle</i> [ <i>parameter_name</i> ]
UnaryOpExpression	" <i>character</i> ..." - use \ to embed "
	UnaryOperator Expression

### Operators

Arithmetic	+ - * /%	Bitwise & ^
Boolean	< <= > >= &&    == !=	Unary + - ~ !

### DataTypes

[ Boolean | Character | String | Real | Integer | GenericValue | Handle | Group<base\_type> | GroupItem<base\_type> | Ref<class\_name> | IncidentHandle | UserDefined enumeration | UserDefined typedef ]

### Builtin Methods

	- invoke via <expression>.<method>(args)
Group	{ <b>addFront</b> (item)   <b>addBack</b> (item)   <b>insert</b> (iter, item)   <b>front</b> ()   <b>back</b> ()   <b>remove</b> (item)   <b>erase</b> (iter)   <b>removeAll</b> ()   <b>size</b> () }
GroupItem	{ <b>current</b> ()   <b>next</b> ()   <b>previous</b> ()   <b>finished</b> ()   <b>front</b> ()   <b>back</b> ()   <b>setGroup</b> (group) }