



---

## **PathMATE Interface Realization Support**

Version 1.4  
March 4, 2004

---

### *PathMATE Technical Notes*

Pathfinder Solutions LLC  
33 Commercial Drive, Suite 2  
Foxboro, MA 02035 USA  
[www.PathfinderMDA.com](http://www.PathfinderMDA.com)  
888-662-7284

# Table Of Contents

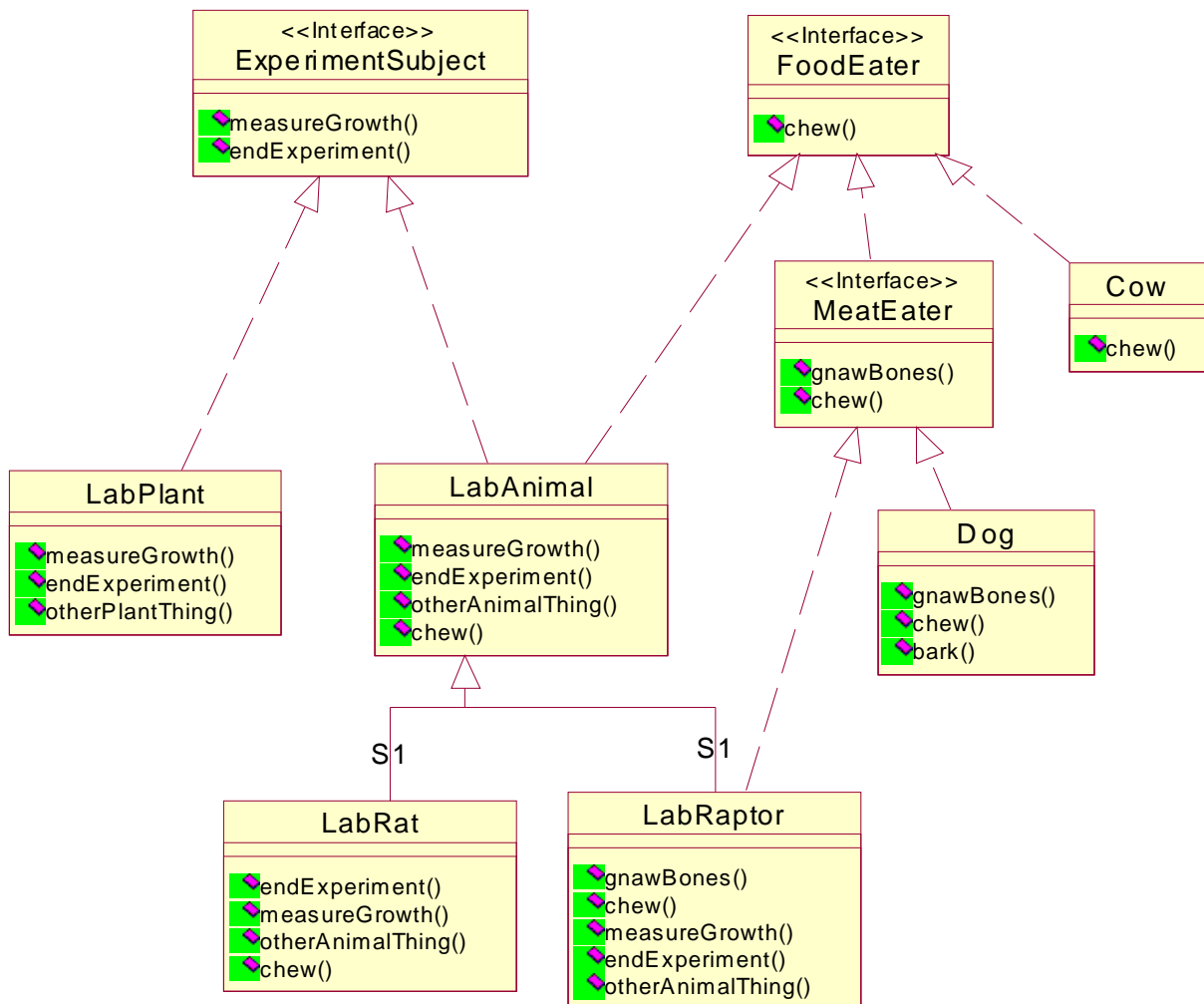
- 1. Introduction..... 1
- 2. Feature Overview..... 1
  - Analysis Constraints .....2
- 3. PathMATE Extract and Transformation Engine Impacts ..... 4
- 4. PathMATE Maps Impacts..... 4
- 5. Constraint Checking ..... 4

## 1. Introduction

This Technical Note describes support to be provided in the PathMATE product family for Interface Realization using the UML Realize relationship. This document outlines modeling conventions and constraints, model checking features, and implementation features.

## 2. Feature Overview

A UML *Interface* class specifies the interfaces for a set of class services (operations). These services may be instance-based or class-based. Zero or more regular UML classes may specify they implement the specified set of services. This is indicated through the use of the UML *Realize* relationship, with the arrow pointing from the *implementing* class to the *Interface* class. In addition, Interface classes can Realize interfaces specified by more general Interface classes.



**Figure 1: Interface Realization class example model**

In Figure 1 the ExperimentSubject Interface class specifies an interface (measureGrowth() and endExperiment()) implemented by the LabPlant and LabAnimal classes. In addition, the LabAnimal implements the FoodEater interface. A specialized form of the FoodEater interface is specified by the MeatEater interface class, and is implemented by the Dog and LabRaptor classes. The LabAnimal shows interface realization by a supertype. LabRaptor shows that a single implementation of chew() is needed for the chew() service inherited from it's partner and also realized from an interface.

The Interface class has a very narrow role – only to specify service profiles. In addition to other limitations outlined below, and Interface class is *abstract* – like a supertype class it cannot be instantiated. However in action language references to an Interface class are allowed. Operations specified by the Interface class may be invoked from an Interface class reference.

The action language fragment below will measure the growth all Lab things:

```
FOREACH thing = CLASS ExperimentSubject
{
    thing.measureGrowth();
}
```

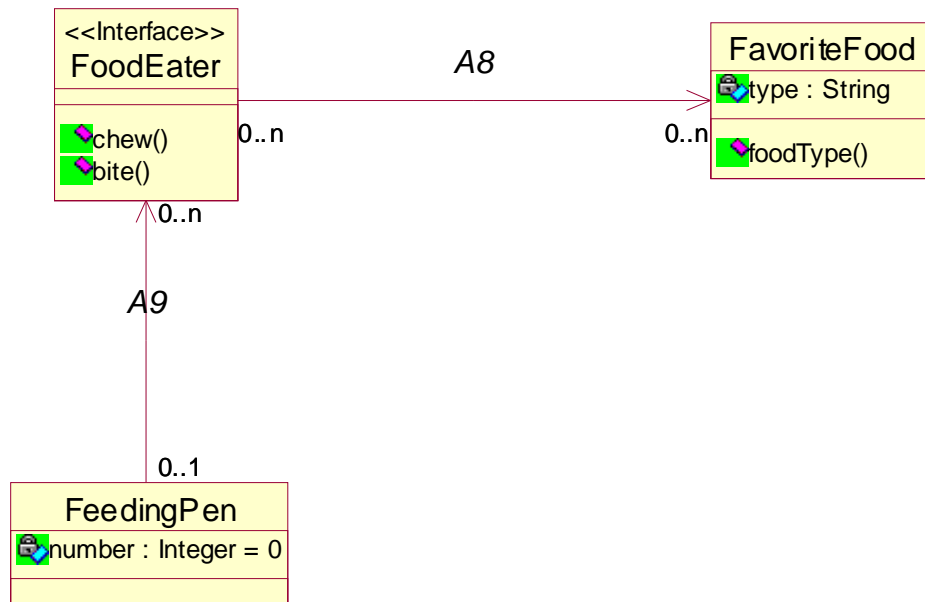
In the above action fragment the measureGrowth() operation ends up being invoked on all instances of LabPlant and LabAnimal.

## Analysis Constraints

An Interface class can only specify operations.

- An Interface class cannot have attributes, states or class-based (static) operations
- An Interface class cannot be an association class
- An Interface class cannot define MBSE events
- An Interface class cannot be instantiated

An interface may participate in associations (See the UML 2.0 Superstructure, p 90). Classes that implement the interface must then present operations that support the association operations.



**Figure 2: Associations and Interfaces**

Class extents, aka instance lists, are kept for all interfaces. FIND and FOREACH are supported for Interface instance lists.

Association navigation to and from interfaces is supported with existing action languagenavigation constructs. Associations between interfaces are also supported.

For example, to link and unlink an interface to a class:

```

Ref<FavoriteFood> food = FIND FavoriteFood;
Ref<FoodEater> eater = FIND Dog; // casts are OK too.
LINK food A8 eater;
Ref<FoodEater> eater2 = FIND food -> A8; // navigation
UNLINK food A8 eater;
  
```

Interfaces may have sub types and super types. An interface may only have other interfaces as supertypes. An interface may have other interfaces as subtypes, or classes as subtypes through the <<realizes>> association.

An implementing class must realize each realized service exactly once:

- Every class that Realizes an Interface class must specify operations matching every operation in the Interface class. An operation from an interface may be implemented in a supertype class.
- If a class realizes or inherits (subtype/supertype) from two or more classes, and two or more of these parents or interfaces have services with the same name, all of these like named inherited/realized services must resolve to the same single root service in the top level Interface. This is shown correctly in the `LabRaptor.chew()` service above, where

it inherits chew() from LabAnimal and realizes chew() from MeatEater, and both of these are defined at the same source: FoodEater.chew().

- If a class inherits/realizes two or more services with the same name and they do NOT come from the same common root Interface service, this is an error.

Methods from an interface may be implemented within an inheritance association at any level of the hierarchy, beneath the class that implements the association. For example, from Figure 1, the measureGrowth() method is defined in ExperimentSubject and implemented in LabAnimal, LabRat, and LabRaptor. Care must be taken here when porting across languages. For example, the Java implementation will always invoke the lowest level class that implements the method (always virtual). In C++, if the methods are not declared virtual (with a property) then the method invoked will depend upon the type of pointer held – LabAnimal or LabRat.

### 3. PathMATE Extract and Transformation Engine Impacts

The PathMATE for Rose extract facility will recognize the Realize relationship and convey that information to the Transformation Engine. The Engine will import the Realize relationship from XMI data, and perform appropriate checking (see Analysis Constraints).

Interface classes must use the <<interface>> stereotype.

### 4. PathMATE Maps Impacts

Java and C++ will support Interface Realization with language support for inheritance. C will support Interface Realization with generated function dispatchers very similar to how operation polymorphism is currently supported for subtype/supertype hierarchies.

### 5. Constraint Checking

The Maps, Extract, and the Transformation Engine check the following constraints on interface usage and insert an error message into the generated file when these constraints fail.

- interfaces cannot have attributes
- interfaces cannot have static methods
- interfaces can only inherit from interfaces, not classes
- a class must implement all the interfaces of all the class it inherits from
- a classes parent class may implement the methods instead
- a class may inherit from at most one class
- name collisions among implemented interfaces