Title:	CIMPLE Provider Development Manual
Author:	Michael E. Brasher
Company:	Inova Development
Date:	February 20, 2006
Version:	1.0
URL:	http://cimple.org/provdevman.pdf

Change History			
Revision	**Date**	**Name**	**Comments**
1.0	2/20/2006	M. Brasher	Initial version

CIMPLE Provider Development Manual

Michael E. Brasher
Inova Development
February 20, 2006

1 Overview

This manual explains how to write providers using CIMPLE. As you will see, CIMPLE makes provider development much easier and the providers you write will be smaller, faster, and more reliable. And with CIMPLE, you can write a provider once and use it with multiple CIM servers, including OpenPegasus and CMPI-compliant servers (OpenWBEM, SFCB, and others).

CIMPLE simplifies provider development in five majors ways: with *first-class objects*, *provider generation*, *operation reduction*, *automated validation*, and *automated registration*. These are described below. Along the way, you will notice many other simplifications as well.

1.1 First-class Objects

CIMPLE generates real C++ classes from MOF class definitions. This offers several advantages over conventional provider interfaces that represent MOF classes with complex dynamic data structures:

- lower development cost.
- reduced code complexity.
- compile-time type checking.
- higher degree of program correctness.
- reduced footprint.
- lower memory usage.
- better performance.

To illustrate class generation, consider the following MOF definition (contained in `Fan.mof`).

```
class Fan
{
    [key] string DeviceID;
    uint64 Speed;
    uint64 DesiredSpeed;
    uint64 SetSpeed([in] uint64 DesiredSpeed);
};
```

Using a tool called `genclass`, we generate the C++ class (and its associated meta-data) like this.

```
% genclass –M Fan.mof Fan
Created Fan.h
Created Fan.cpp
```

Here is the generated C++ class definition (for a full listing see Appendix A).

```
class Fan : public Instance
{
public:
    Property<String> DeviceID;
    Property<uint64> Speed;
    Property<uint64> DesiredSpeed;
    CIMPLE_CLASS(Fan)
};
```

Having concrete classes means detecting many errors at compile time, rather than run time. Concrete classes are subject to the static type checking facilities of the compiler. We will say much more about first-class objects and `genclass` later on.

1.2 Provider Generation

CIMPLE provides a tool called `genprov` that generates provider skeleton code automatically. It generates stubs for the intrinsic CIM operations (e.g., get-instance, enum-instance) as well as a stub for each extrinsic method in the MOF class. With CIMPLE, extrinsic methods are ordinary C++ member functions. For example, the following command generates a provider from the `Fan` definition above.

```
% genprov –M Fan.mof Fan
Created Fan_Provider.h
Created Fan_Provider.cpp
```

Here is the stub that is generated for the extrinsic get-instance operation.

```
Get_Instance_Status Fan_Provider::get_instance(
    const Fan* model,
    Fan*& instance)
{
    return GET_INSTANCE_UNSUPPORTED;
}
```

And here is the stub that is generated for the `SetSpeed` extrinsic method.

```
Invoke_Method_Status Fan_Provider::SetSpeed(
    const Fan* instance,
    const Property<uint64>& DesiredSpeed,
    Property<uint64>& return_value)
{
   return INVOKE_METHOD_UNSUPPORTED;
}
```

Later, we will discuss how these are used to implement providers.

1.3 Operation Reduction

CIMPLE reduces the number of operations you must implement. Most operations are implemented for you, in terms of other operations. For example, `enum-instance-names` is treated as special case of `enum-instances`, in which only the key properties are requested. The table below shows all the operations and indicates which ones you are required to implement.

Operation	**Required**	**Notes**
`get-instance`	Maybe	CIMPLE may implement through enum-instances.
`enum-instance`	Yes	
`enum-instance-names`	No	Implemented in terms of enum-instance (with filtering).
`get-property`	No	Implemented in terms of get-instance (with filtering).
`create-instance`	Maybe	Needed only by non-read-only providers.
`modify-instance`	Maybe	Needed only by non-read-only providers.
`set-property`	No	Implemented in terms of modify-instance (with filtering).
`delete-instance`	Maybe	Needed only by non-read-only providers.
`associators`	No	Implemented in terms of enum-instance and get-instance.
`associator-names`	No	Implemented in terms of enum-instance.
`references`	No	Implemented in terms of enum-instance.
`reference-names`	No	Implemented in terms of enum-instance.
`enable-indications`	Yes	Needed only by indication providers.
`disable-indications`	Yes	Needed only by indication providers.
`invoke-method`	No	Automatically generated by `genprov`.
`Method-stubs`	Yes	Needed only by method providers.

Most operations are implemented for you or are not needed. In practice, most providers only need to implement one or two operations. Later we will discuss how to implement these operations.

1.4 Automated Validation

CIMPLE provides a tool called `testprov` that automatically loads and tests provider modules. The purpose of the tool is find any errors in providers prior to provider registration. The testprov tool currently checks the following for each provider found in the module.

- Whether the library can be dynamically loaded.
- Whether the provider can be loaded.
- Whether any instances can be obtained with the enum-instances operation.
- Whether each instance obtained with enum-instances can be obtained with get-instance.
- Whether instances obtained with enum-instances are identical to those obtained with get-instance.
- Whether an instance can be modified successfully.
- Whether an instance can be deleted successfully.
- Whether the provider can be unloaded.

Further, testprov provides an isolated environment for finding memory leaks when used with tools like Valgrind or Purify. Leaks are not as easily found in the context of a CIM server.

Running testprov is easy, for example:

```
% testprov libcmplfan.so
```

We will explain how to use testprov later.

1.5 Automated Registration

One of the difficulties of developing providers is registration, which involves configuring a CIM server to load and use your provider. There are at least three steps:

1. Adding any user-defined classes to the CIM-server repository.
2. Modifying the CIM-server configuration to recognize your provider.
3. Moving the provider module into a location where the CIM server can find and load it.

CIMPLE provides a tool called `regmod`, which automatically registers all the providers in a module with the OpenPegasus server, with a single command. Although `regmod` is not available yet for other CIM servers, you can always register CIMPLE providers manually in those environments (which you would have to do in any case).

Once a provider module is developed, registering all of its provider is trivial. For example, to register our `Fan` provider, we would simply type this.

```
% regmod libcmplfan.so
```

While registering the provider module (libcmplfan.so) with the OpenPegasus server, `regmod` does the following.

- Validates the OpenPegasus environment.
- Verifies that the OpenPegasus repository exists and is valid.
- Verifies that OpenPegasus contains the CIMPLE provider manager and patch.
- Verifies that each provider can be loaded and unloaded.
- Adds all provider-defined CIM classes to the OpenPegasus repository.
- Checks whether provider-defined CIM classes are compatible with OpenPegasus versions.
- Creates the necessary provider registration instances in the OpenPegasus repository.
- Copies the module into the OpenPegasus environment.

Registration is a tedious and error-prone process. With `regmod`, it is trivial and only takes a few seconds.

2 The CIM Data Types

This chapter introduces the CIM data types, which are used to build CIM classes (discussed in the next chapter). The following table shows the basic CIM data types and their C++ representations.

CIM Data Type	**CIMPLE Type Name**	**C++ Representation**
boolean	`boolean`	`bool`
uint8	`uint8`	`unsigned char`
sint8	`sint8`	`signed char`
uint16	`uint16`	`unsigned short`
sint16	`sint16`	`signed short`
uint32	`uint32`	`unsigned int`
sint32	`sint32`	`signed int`
uint64	`uint64`	`unsigned long long (GCC)`
sint64	`sint64`	`signed long long (GCC)`
real32	`real32`	`float`
real64	`real64`	`double`
char16	`char16`	`unsigned short`
string	`String`	`String class`
datetime	`Datetime`	`Datetime class`

Most of the CIM data types can be represented with primitive C++ types, except for string and datetime. CIMPLE provides a `String` class and `Datetime` class to represent these as well as an `Array` class for defining arrays of CIM data types. The `String`, `Datetime`, and `Array` classes are discussed in the following sections.

2.1 The `String` Class

The `String` class provides a convenient interface for defining UTF8 character sequences. It was designed to be efficient and small. It employs fast atomic reference counting and only consumes a little over two kilobytes of object code. The following example builds a path from a directory name and file name.

```
const char* dn;
const char* fn;
…
String path(dn);
path.append(‘/’);
path.append(fn);
```

You will find the `String` class similar to other string classes except operations have been omitted that lend themselves to poor performance or code bloat. For example, the notorious `operator+()` function is not supported. Consider the two implementations below, using the standard `string` class.

```
path = string(fn) + ‘/’ + string(dn);
```

```
path.append(dn);
path += ‘/’;
path.append(fn);
```

The implementation on the left (based on the `operator+()` function) turned out to be 2.5 times slower and produced 3.5 times more object code.

We hope you will find using the `String` class relatively straightforward. For details about its interface see the `<cimple/String.h>` header file.

2.2 The `Datetime` Class

The `Datetime` class is used to represent either a *timestamp* or an *interval*. It encapsulates two fields whose meaning is given in the table below.

Field	**Timestamp**	**Interval**
`uint64 usec`	Microseconds elapsed since the epoch: January 1, 1970 00:00:00	Microseconds elapsed since an arbitrary point in time
`sint64 offset`	UTC offset in minutes	Always 0xFFFFFFFFFFFFFFFF

The following code snippet initializes a timestamp from a microseconds and an offset component.

```
Datetime timestamp(usec, offset);
```

And this snippet initializes an interval from a microseconds component (the offset field is set to 0xFFFFFFFFFFFFFFFF).

```
Datetime interval(usec);
```

A `Datetime` object may also be initialized from a canonical CIM string representation. For example, to initialize a timestamp:

```
// January 1, 2006 12:30:00 UTC+360
Datetime timestamp;
timestamp.set(“20060101123000000000+360”);
```

Or to initialize an interval:

```
// 1 day, 2 hours, 3 minutes, 4 seconds, 5 microseconds
Datetime interval;
interval.set(“00000001020304.000005:000”);
```

See the CIM infrastructure specification for more information on CIM datetime strings. The following table describes a few of the key member functions.

Member function	**Description**
`Datetime::now()`	Creates a timestamp representing the current time.
`Datetime::is_timestamp()`	Returns true if the `Datetime` object represents a timestamp.
`Datetime::is_interval()`	Returns true if the `Datetime` object represents an interval.
`Datetime::ascii()`	Obtains an ASCII representation of the `Datetime` object, in canonical CIM string representation.

As with the `String` class, `Datetime` is efficient and small. It is represented with two 64-bit integers so assignment and copying is very fast and `Datetime.o` is less than two kilobytes.

For more details, see the `<cimple/Datetime.h>` header file.

2.3 The `Array<>` class

`Array<>` is a class for building dynamic arrays of CIM data types. Although it is a template, it uses special techniques to eliminate virtually all code bloat ordinarily attributed to templates. The object size of the base implementation is under two kilobytes.

It is similar to other array classes you may have used. It provides methods for adding, getting, setting, and removing elements. For example, the following snippet defines a zero-size array of uint32.

```
Array<uint32> a;
```

CIMPLE also provides a `typedef` of `Array<uint32>` called `Array_uint32`. You may use these interchangeably. The following appends two elements to the array.

```
a.append(100);
a.append(200);
```

And finally, the following iterates the array, printing out each element.

```
for (size_t i = 0; i < a.size(); i++)
    printf("%u\n", a[i]);
```

As you might expect, arrays can be assigned and copied. For example:

```
Array<uint32> b = a;
Array<uint32> c;
c = b;
```

You may also define arrays of objects too. For example, the following defines an array of `String` objects and appends two strings.

```
Array<String> a;
a.append("one");
a.append("two");
```

To learn more about the `Array<>` class see the `<cimple/Array.h>` header.

3 Generating and Using CIM Classes

This chapter explains how to generate and use classes. We show how to use the `genclass` tool and how to create, initialize, and destroy classes.

3.1 Generating A Simple Class

We will start by generating a simple class called `Person` from the following MOF definition, contained in `repository.mof`.

```
class Person
{
    [key] uint32 Id;
    string LastName;
    char16 MiddleInitial;
    string FirstName;
    uint16 Age;
};
```

Next we use `genclass` to translate this MOF definition into a C++ class (for help using this command type "`genclass –h`").

```
% genclass Person
Created Person.h
Created Person.cpp
```

`Person.h` contains the C++ class definition, shown below. `Person.cpp` contains meta data used internally by CIMPLE (not used by provider developers). The C++ class looks similar to the MOF class.

```
class Person : public Instance
{
public:
    // Person features:
    Property<uint32> Id;
    Property<String> LastName;
    Property<char16> MiddleInitial;
    Property<String> FirstName;
    Property<uint16> Age;
    CIMPLE_CLASS(Person)
};
```

But there are two important differences. First, properties are represented using the `Property<>` template. And second, all generated CIMPLE classes derive from `Instance` rather than their CIM super class. These differences are explained in the next two sections.

3.2 The `Property<>` Structure

As we mentioned before, generated classes use the `Property<>` template to define their properties. This is necessary to implement the CIM concept of `NULL`. The `Property<>` structure is defined as follows.

```
template<class T>
struct Property
{
   uint8 null;
   T value;
};
```

The following illustrates how to set a non-null property.

```
Property<uint32> Age;
Age.value = 21;
Age.null = false;
```

And this sets a property to `NULL`.

```
Property<uint32> Age;
Age.null = false;
```

The value is not important since it is ignored when the `null` flag is set. You will probably never define a `Property<>` object directly as shown in the examples above. But you will use `Property<>` objects as part of generated classes, which we discuss later on.

3.3 The CIMPLE Inheritance Model

This section explains how CIMPLE supports CIM class inheritance, which regrettably cannot be implemented directly with C++ inheritance due to CIM reference overriding (explained shortly). We will start with an example. Take the MOF class definitions below.

```
class Class1
{
   uint32 x;
};

class Class2 : Class1
{
   string y;
};

class Class3 : Class2
{
   boolean z;
};
```

From these, `genclass` produces the following C++ definition for `Class3`.

```
class Class3 : public Instance
{
    // Class1 features:
   uint32 x;
   // Class2 features:
   string y;
   // Class3 features:
   boolean z;
   CIMPLE_CLASS(Class3);
};
```

You will notice two things about the generated class. First, it inherits from `Instance` rather than `Class2`. Second, it includes inherited properties `x` and `y`. CIMPLE flattens out the inheritance hierarchy in this way by including all inherited features in the order in which they were inherited. This approach properly supports *polymorphism*, that is an instance of `Class3` can be substituted wherever an instance of `Class1` or `Class2` is needed. We demonstrate this in the following example.

```
void print_Class1(Class1* class1)
{
    printf(“%u\n”, class1->x.value);
}

void print_Class2(Class2* class2)
{
    print((Class1*)class2);
    printf(“%s\n”, class2->y.value.c_str());
}
```

Notice how we pass an instance of `Class2` to `print_Class1()`, which expects an instance of `Class1`. This works because the properties that `Class2` inherits from `Class1` come first and are in the same order. This makes any instance of `Class2` structurally compatible with `Class1` but not visa versa of course (since `Class2` may declare additional properties). It is easy to see how this scheme generalizes to multiple levels of inheritance.

3.4 The `cast` operator

In the last example we *statically casted* from `Class2` to `Class1` because we knew the instance was in fact an instance of `Class1`. CIMPLE also supports *dynamic casting* with the `cast` operator, which returns zero if its argument is not an instance of the desired type. For example:

```
Class2* class2 = Class2::create();
Class1* class1 = cast<Class1*>(class2);

if (class1)
{
    // class2 must be an instance of Class1.
}
```

This operator allows you to check at run time whether something is an instance of a given class. The example above demonstrates “up casting” (i.e., casting up the inheritance tree). The `cast` operator can also do “down casting” (i.e., casting down the inheritance tree). For example:

```
void f(Class1* class1)
{
    Class2* class2 = cast<Class2*>(class1);

    if (class2)
    {
        // class1 is an instance of Class2.
    }
}
```

The cast operator supports *dynamic casting* (resolved at run time), whereas *static casting* is resolved at compile time. To improve performance and reduce object code size, use static casting when you can. Dynamic casting is slower and produces more code but sometimes it is necessary.

3.5 The `is_a` operator

The is_a operator determines, at run time, whether an instance is of a given class. For example, the following expression returns true if `inst` is and instance of `Class3`.

```
is_a<Class3>(inst)
```

If this operator returns true, it is safe to treat `inst` as an instance of Class3 (e.g., statically cast it). Since you can also check is-a relationships with the `cast` operator, the `is_a` operator is not used very often.

3.6 Working with CIM Instances

Continuing with the our `Person` example, we show how to use a generated CIM class. First we create an instance of `Person`.

```
Person* p = Person::create();
```

`Person::create()`allocates a Person instance and initializes all properties (by default properties are non-null). Next we initialize our instance.

```
p->Id.value = 444556666;
p->LastName.value = “Brasher”;
p->MiddleInitial = ‘E’;
p->FirstName.value = “Michael”;
p->Age.null = true;
```

And now we print the instance.

```
print(p);
```

Which makes this appear on standard output.

```
Person
{
    uint32 Id = 444556666;
    string LastName = "Brasher";
    char16 MiddleInitial = ‘E’;
    string FirstName = "Michael";
    uint16 Age = null;
}
```

Along the way we might want to make a copy of the object like this.

```
Person* pc = clone(p);
```

And eventually, all instances we create should be passed to `destroy()` to release memory.

```
destroy(p);
destroy(pc);
```

As you can see, working with instances is relatively straightforward.

3.7 Special Instance Functions

This table below describes some special functions for manipulating instances (defined in `<cimple/Instance.h>`). Uses of `i`, `i1`, or `i2` in the definitions below refer to instances.

Function	**Description**
`identical(i1,i2)`	Returns true if the two instances are identical (i.e., are instances of the same class and have the same property values).
`key_eq(i1, i2)`	Returns true if the two instances are key equivalent (i.e., their key components are identical).
`copy(i1, i2)`	Copies all class features from instance `i2` to instance `i1`.
`copy_keys(i1, i2)`	Copies all key features from instance `i2` to instance `i1`.
`clone(i)`	Creates a new instance that is an exact copy of instance `i`.

`key_clone(i)`	Create a new instance whose key features are copied from `i`, but whose other features are initialized to null.
`nullify_properties(i)`	Sets all properties in the instance to null.
`de_nullify_properties(i)`	Sets all properties in the instance to non-null.
`nullify_keys(i)`	Sets all key properties to null.
`de_nullify_keys(i)`	Sets all key properties to non-null.
`nullify_non_keys(i)`	Set all non-key properties to null.

These functions are defined in `<cimple/Instance.h>`.

3.8 Working With References

<coming soon>

3.9 Working With Embedded Instances

<coming soon>

4 A First Provider

This chapter shows how to write a CIMPLE instance and method provider for the `Fan` class presented earlier.

4.1 Generating the `Fan` Provider

The first step is to automatically generate the `Fan` provider with the `genprov` command. Recall the MOF definition for Fan shown earlier (contained in `Fan.mof`).

```
class Fan
{
    [key] string DeviceID;
    uint64 Speed;
    uint64 DesiredSpeed;
    uint64 SetSpeed([in] uint64 DesiredSpeed);
};
```

We already generated the C++ class. Now we show how to generate the provider skeleton code. We do this with the genclass command as follows.

```
% genprov –M Fan.mof Fan
Created Fan_Provider.h
Created Fan_Provider.cpp
```

Genprov generates two files: `Fan_Provider.h` and `Fan_Provider.cpp`. `Fan_Provider.h` contains the interface for the `Fan_Provider` class. Fan_Provider.cpp defines the member functions or stubs that we will implement below. For a full listing see Appendix B.

4.2 Implementing `Fan_Provider::enum_instances()`

First we implement `enum_instances()`. Genprov produced the following stub.

```
Enum_Instances_Status Fan_Provider::enum_instances(
    const Fan* model,
    Enum_Instances_Handler<Fan>* handler)
{
    return ENUM_INSTANCES_OK;
}
```

This function has two parameters: `model` and `handler`. The `model` parameter is an instance of the `Fan` class. It tells us which properties have been requested. For example, the following snippet checks whether the `Speed` property must be provided.

```
if (!model->Speed.null)
{
    // The Speed property has been requested.
}
```

We may safely choose to ignore the `model` parameter and just provide all properties. But some properties are expensive to obtain so we recommend using the model to improve performance. The `handler` parameter contains a callback used to pass enumerated instances to the requestor.

Now we implement `enum_instances()`. The implementation, shown below, creates and passes one `Fan` instance to the requestor.

```
Enum_Instances_Status Fan_Provider::enum_instances(
    const Fan* model,
    Enum_Instances_Handler<Fan>* handler)
{
    Fan* fan1 = Fan::create();
    fan1->DeviceID.value = “FAN1”;

    if (!model->Speed.null)
        fan1->Speed.value = 10000;

    if (!model->DesiredSpeed.null)
        fan1->DesiredSpeed.value = 10000;

    handler->handle(fan1);

    return ENUM_INSTANCES_OK;
}
```

Two things should be noted. First, it is unnecessary to check `model->DeviceID.null` since it is a key and keys must be provided (otherwise an instance would have no identity). Second, the requestor is responsible for releasing instances passed to `handle()`, not the provider. The provider developer must be careful not to call `destroy()` on any instances passed to `handle()`.

4.3 Implementing `Fan_Provider::get_instance()`

Next we show how to implement `Fan_Provider::get_instance()`. Here is the stub produced by `genprov`.

```
Get_Instance_Status Fan_Provider::get_instance(
    const Fan* model,
    Fan*& instance)
{
    return GET_INSTANCE_UNSUPPORTED;
}
```

As it stands, this is a working implementation. CIMPLE first calls `get_instance()`; if `get_instance()` returns `GET_INSTANCE_UNSUPPORTED`, CIMPLE then calls `enum_instance()` to find a matching instance. This is adequate for providers whose instances are few and cheap to build. But for performance-critical providers, we must provide a complete implementation.

The `get_instance()` method has two parameters. The `model` parameter indicates which properties must be provided (as discussed before) and it uniquely identifies the target instance, that is CIMPLE initializes its key properties before calling `get_instance()`. The `instance` parameter (initially null) is an output parameter that will point to the newly created instance upon successful return.

The implementation below provides a single hard-coded instance, whose CIM instance name is `Fan.DeviceID=”FAN1”`).

```
Get_Instance_Status Fan_Provider::get_instance(
   const Fan* model,
   Fan*& instance)
{
   if (model->DeviceID.value == “FAN1”)
   {
      if (!model->Speed.null)
         instance->Speed.value = 10000;

      if (!model->DesiredSpeed.null)
         instance->DesiredSpeed.value = 10000;

      return GET_INSTANCE_OK;
   }

   return GET_INSTANCE_NOT_FOUND;
}
```

Since this provider only provides a single instance (`Fan.DeviceId=“FAN1”`), our implementation just produces the instance and returns `GET_INSTANCE_OK`. If any other instance is requested, it returns `GET_INSTANCE_NOT_FOUND`.

4.4 Implementing `Fan_Provider::SetSpeed()`

Implementing the `Fan_Provider::SetSpeed()` method is easy. Recall the MOF definition for the `Fan` class.

```
class Fan
{
    [key] string DeviceID;
    uint64 Speed;
    uint64 DesiredSpeed;
    uint64 SetSpeed([in] uint64 DesiredSpeed);
};
```

From this definition, `genprov` generates the following stub.

```
Invoke_Method_Status Fan_Provider::SetSpeed(
   const Fan* self,
   const Property<uint64>& DesiredSpeed,
   Property<uint32>& return_value)
{
   return INVOKE_METHOD_UNSUPPORTED;
}
```

`SetSpeed()` has three parameters. The `self` parameter identifies the instance whose speed is to be set (this parameter is omitted for static methods). The `DesiredSpeed` parameter is an input parameter, given by the MOF definition. Finally, `return_value` is the return value of the `SetSpeed()` method (it is not possible for it to be the actual function return value since that is needed to return the status. Implementing `SetSpeed()` is trivial so no implementation is shown here.

5 Writing an Association Provider

<coming soon>

6 Writing and Indication Provider

<coming soon>

7 Registering a CMPI Provider

<coming soon>

Appendix A – Generated Fan Class

Fan.h

```
//==============================================================================
//
// PLEASE DO NOT EDIT THIS FILE; IT WAS AUTOMATICALLY GENERATED BY GENCLASS.
//
//==============================================================================

#ifndef _Fan_h
#define _Fan_h

#include <cimple/cimple.h>

CIMPLE_NAMESPACE_BEGIN

class CIMPLE_LINKAGE Fan : public Instance
{
public:
    // Fan features:
    Property<String> DeviceID;
    Property<uint64> Speed;
    Property<uint64> DesiredSpeed;

    CIMPLE_CLASS(Fan)
};

class CIMPLE_LINKAGE Fan_SetSpeed_method : public Instance
{
public:
    Property<uint64> DesiredSpeed;
    Property<uint32> return_value;
    CIMPLE_METHOD(Fan_SetSpeed_method)
};

CIMPLE_NAMESPACE_END

#endif /* _Fan_h */
```

Fan.cpp

```
//==============================================================================
//
// PLEASE DO NOT EDIT THIS FILE; IT WAS AUTOMATICALLY GENERATED BY GENCLASS.
//
//==============================================================================

#include <cimple/Meta_Class.h>
#include <cimple/Meta_Property.h>
#include <cimple/Meta_Reference.h>
#include "Fan.h"

CIMPLE_NAMESPACE_BEGIN

using namespace cimple;

extern CIMPLE_HIDE const Meta_Property _Fan_DeviceID;

const Meta_Property _Fan_DeviceID =
{
    CIMPLE_FLAG_PROPERTY|CIMPLE_FLAG_KEY,
    "DeviceID",
    STRING,
    0,
    CIMPLE_OFF(Fan,DeviceID)
};

extern CIMPLE_HIDE const Meta_Property _Fan_Speed;
```

```
const Meta_Property _Fan_Speed =
{
    CIMPLE_FLAG_PROPERTY,
    "Speed",
    UINT64,
    0,
    CIMPLE_OFF(Fan,Speed)
};

extern CIMPLE_HIDE const Meta_Property _Fan_DesiredSpeed;

const Meta_Property _Fan_DesiredSpeed =
{
    CIMPLE_FLAG_PROPERTY,
    "DesiredSpeed",
    UINT64,
    0,
    CIMPLE_OFF(Fan,DesiredSpeed)
};

static const Meta_Property _Fan_SetSpeed_DesiredSpeed =
{
    CIMPLE_FLAG_PROPERTY|CIMPLE_FLAG_IN,
    "DesiredSpeed",
    UINT64,
    0,
    CIMPLE_OFF(Fan_SetSpeed_method,DesiredSpeed)
};

static const Meta_Property _Fan_SetSpeed_return_value =
{
    CIMPLE_FLAG_PROPERTY|CIMPLE_FLAG_OUT,
    "return_value",
    UINT32,
    0,
    CIMPLE_OFF(Fan_SetSpeed_method,return_value)
};

static Meta_Feature* _Fan_SetSpeed_meta_features[] =
{
    (Meta_Feature*)&_Fan_SetSpeed_DesiredSpeed,
    (Meta_Feature*)&_Fan_SetSpeed_return_value
};

const Meta_Method Fan_SetSpeed_method::static_meta_class =
{
    CIMPLE_FLAG_METHOD,
    "SetSpeed",
    _Fan_SetSpeed_meta_features,
    CIMPLE_ARRAY_SIZE(_Fan_SetSpeed_meta_features),
    sizeof(Fan_SetSpeed_method),
    UINT32,
};

static Meta_Feature* _Fan_meta_features[] =
{
    (Meta_Feature*)&_Fan_DeviceID,
    (Meta_Feature*)&_Fan_Speed,
    (Meta_Feature*)&_Fan_DesiredSpeed,
    (Meta_Feature*)&Fan_SetSpeed_method::static_meta_class,
};

const Meta_Class Fan::static_meta_class =
{
    CIMPLE_FLAG_CLASS,
    "Fan",
    _Fan_meta_features,
    CIMPLE_ARRAY_SIZE(_Fan_meta_features),
    sizeof(Fan),
    0,
    1,
    0xF0EDE346,
};

CIMPLE_NAMESPACE_END
```

Appendix B – Generated Fan Provider Skeleton

Fan_Provider.h

```
#ifndef _Fan_Provider_h
#define _Fan_Provider_h

#include <cimple/cimple.h>
#include "Fan.h"

CIMPLE_NAMESPACE_BEGIN

class Fan_Provider
{
public:

    typedef Fan Class;

    Fan_Provider();

    CIMPLE_HIDE ~Fan_Provider();

    CIMPLE_HIDE Load_Status load();

    CIMPLE_HIDE Unload_Status unload();

    CIMPLE_HIDE Timer_Status timer(uint64& timeout_msec);

    CIMPLE_HIDE Get_Instance_Status get_instance(
        const Fan* model,
        Fan*& instance);

    CIMPLE_HIDE Enum_Instances_Status enum_instances(
        const Fan* model,
        Enum_Instances_Handler<Fan>* handler);

    CIMPLE_HIDE Create_Instance_Status create_instance(
        const Fan* instance);

    CIMPLE_HIDE Delete_Instance_Status delete_instance(
        const Fan* instance);

    CIMPLE_HIDE Modify_Instance_Status modify_instance(
        const Fan* instance);

    CIMPLE_HIDE Invoke_Method_Status SetSpeed(
        const Fan* self,
        const Property<uint64>& DesiredSpeed,
        Property<uint32>& return_value);

    static CIMPLE_HIDE int proc(
        int operation, void* arg0, void* arg1, void* arg2, void* arg3);
};

CIMPLE_NAMESPACE_END

#endif /* _Fan_Provider_h */
```

Fan_Provider.cpp

```
#include "Fan_Provider.h"

CIMPLE_NAMESPACE_BEGIN

Fan_Provider::Fan_Provider()
{
}
```

```
Fan_Provider::~Fan_Provider()
{
}

Load_Status Fan_Provider::load()
{
    return LOAD_OK;
}

Unload_Status Fan_Provider::unload()
{
    return UNLOAD_OK;
}

Timer_Status Fan_Provider::timer(uint64& timeout_msec)
{
    return TIMER_CANCEL;
}

Get_Instance_Status Fan_Provider::get_instance(
    const Fan* model,
    Fan*& instance)
{
    return GET_INSTANCE_UNSUPPORTED;
}

Enum_Instances_Status Fan_Provider::enum_instances(
    const Fan* model,
    Enum_Instances_Handler<Fan>* handler)
{
    return ENUM_INSTANCES_OK;
}

Create_Instance_Status Fan_Provider::create_instance(
    const Fan* instance)
{
    return CREATE_INSTANCE_UNSUPPORTED;
}

Delete_Instance_Status Fan_Provider::delete_instance(
    const Fan* instance)
{
    return DELETE_INSTANCE_UNSUPPORTED;
}

Modify_Instance_Status Fan_Provider::modify_instance(
    const Fan* instance)
{
    return MODIFY_INSTANCE_UNSUPPORTED;
}

Invoke_Method_Status Fan_Provider::SetSpeed(
    const Fan* self,
    const Property<uint64>& DesiredSpeed,
    Property<uint32>& return_value)
{
    return INVOKE_METHOD_UNSUPPORTED;
}

int Fan_Provider::proc(
    int operation, void* arg0, void* arg1, void* arg2, void* arg3)
{
    // CAUTION: PLEASE DO NOT MODIFY THIS FUNCTION; IT WAS AUTOMATICALLY
    // GENERATED.

    typedef Fan Class;
    typedef Fan_Provider Provider;

    if (operation != OPERATION_INVOKE_METHOD)
        return Provider_Proc_T<Provider>::proc(
            operation, arg0, arg1, arg2, arg3);
    Provider* provider = (Provider*)arg0;
    const Class* self = (const Class*)arg1;
    const char* meth_name = ((Instance*)arg2)->meta_class->name;

    if (strcasecmp(meth_name, "SetSpeed") == 0)
    {
        typedef Fan_SetSpeed_method Method;
        Method* method = (Method*)arg2;
        return provider->SetSpeed(
            self,
            method->DesiredSpeed,
            method->return_value);
    }
    return -1;
}

CIMPLE_NAMESPACE_END
```